DIGITAL TWIN PLATFORM FOR DRONE APPLICATIONS

AMAL DEV HARIDEVAN

A THESIS SUBMITTED TO THE FACULTY OF GRADUATE STUDIES IN
PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

GRADUATE PROGRAM IN
EARTH AND SPACE SCIENCE
YORK UNIVERSITY
TORONTO, ONTARIO

JULY 2025

Abstract

Simulation tools and digital twins are essential to UAV research, supporting rapid testing, prototyping, and deployment of algorithms in control, computer vision, and deep learning. State-of-the-art simulators lack modularity, scalability, and cross-disciplinary integration, while the persistent sim-to-real gap limits the transfer of algorithms to hardware. Despite these challenges, simulators remain indispensable, particularly for deep reinforcement learning (DRL), where large-scale training in the real world is impractical due to safety and hardware constraints.

This thesis presents a modular, scalable, and transferable simulation platform for quadrotor UAVs, emphasizing high-fidelity dynamics modeling, efficient control design, and seamless integration with embodied AI research. A digital twin based on Blade Element Theory is developed for the Quanser QDrone2, with parameterized forces enabling extension to other architectures. A Control library within the Gazebo ecosystem accelerates controller prototyping and benchmarking. Gazebo plugins and ROS1/ROS2 integration reduce computational overhead and support direct transfer of algorithms from simulation to hardware. To advance DRL research, deterministic and stochastic environments are introduced, enabling reproducibility, realistic modeling of uncertainties, and large-scale parallel training. Experimental validation confirms the platform's fidelity and effectiveness in bridging simulation and real-world UAV deployment.

Dedication

To my parents, my brother, and professor

Acknowledgments

I would like to express my sincere gratitude to my supervisor, Prof. Jinjun Shan, for giving me the opportunity to be his student. I am forever grateful to Professor Shan for his support, his patience, and his knowledge. I was able to hone my skills, learn my limitations, and enhance my strengths under his supervision. If not for his patience and understanding, I wouldn't be the person I am today. Thanks for being an inspiration, for showing me the path forward, and teaching me values. I would also like to thank my committee member, Prof. Michael Bazzochi, for providing insights into my work during our invaluable meetings. I would like to give special thanks to Dr Junjie Kang for providing valuable insights during my journey. His knowledge and guidance were pivotal in my personal and academic development. Meeting Dr. Yibo Liu was a critical event in my life, without which I wouldn't be writing this. Thanks for providing the insights and motivation you gave me. Thanks to Dr. Mingfeng Yuan for sharing his valuable wisdom with me, which I still recall to this day. Thanks to Hunter for being a strong influence and inspiration, and for his valuable knowledge. I would like to express my gratitude to Hao Wang for his advice and inspiration that he gave me during my first year, which was an essential part of my journey. I would like to thank my friends and colleagues at SDCNLab- Reza, Dr. Xiaoyu Zhu, Pukun, Yida, Ali, Ahmed, Larissa and Hao Zhang for their valuable support and kindness that they have shown me during my time here.

Finally, I am grateful to my family, my parents, and my brother, for being in my life, for enabling me to accomplish a checkpoint in this journey. Without the sacrifices they made, I would not be here today, and I am forever in debt to them.

Table of Contents

List of Tables

List of Figures

Nomenclature

Topology

$GL(3)$	General linear group
S^3	Unit sphere in $\mathbb{R}^4$
$SE(3)$	Lie group
$SO(3)$	Special orthogonal group

Frames of Reference

$\mathbf{e_i}$	Unit vector along axis i
${}^{\mathbf{I}}\mathbf{e}_{\mathbf{i}}^{\mathbf{B}}$	Unit vector along axis i in body fixed frame, expressed in inertial frame

Functions

$[.]_\times$	Skew-symmetric matrix transformation of a vector
$\dot{(.)}$	Time derivative of a physical quantity
$\hat{(.)}$	Hat map

$\odot$	Hadamard product
π_θ	Stochastic policy
diag(.)	Diagonal matrix with elements from argument
Q_{π_θ}	Action value function parametrized with parameters θ
V_{π_θ}	State value function parametrized with parameters θ
$(.)^\vee$	Vee map, inverse of hat map, transforms a skew-symmetric matrix to vector

Scalars

$\gamma_1,\ \gamma_2$	Swinging angles for tethered payload
$\omega_{i,\ rotor}$	Angular velocity of rotor i
$\phi,\ \theta,\ \psi$	Euler angles for the sequential rotations along $\mathbf{e_1}$, $\mathbf{e_2}$, and $\mathbf{e_3}$
${}^B(.)$	Element of a vector projected along an axis in Body fixed frame B
${}^I(.)$	Element of a vector projected along an axis in Inertial frame I
$f_{control}$	Control frequency
$f_{physics}$	Physics frequency
$L_\phi,\ L_\theta$	Roll and pitch moment arms
RTF	Real time factor for simulation
$T_{physics}$	Physics step size

Vectors

$\hat{\mathbf{l}}$	Unit vector for a tethered payload

$\mathbf{0}^N$	Zero vector of shape N
$\overrightarrow{\mathbf{I}\mathbf{g}}$	Acceleration due to gravity in inertial frame
X	State vector

Matrices

$\mathbf{I}^{n\times n}$	Identity matrix of shape n
$\mathcal{A}^I$	Aerodynamic drag matrix
A	Thrust to generalized force mapping matrix
J	Principal inertia matrix
J_{rotor}	Moment of inertia of propellor
${}^B_I R$	Rotation matrix from inertial frame to body fixed frame
${}' A^{-1}$	Generalized forces to rotor velocities mapping matrix

List of Acronyms

LiDAR	Light Detection and Ranging
6-DOF	6 Degrees Of Freedom
GPS	Global Positioning System
SLAM	Simultaneous Localization and Mapping
RGB-D	Red Green Blue-Depth
IMU	Inertial Measurement Unit
ROS	Robot Operating System
FoV	Field of View
MoCap	OptiTrack Motion Capture
CPU	Central Processing Unit
RMSE	Root-mean-square error
GPU	Graphics Processing Unit
ODE	Ordinary Differential Equation
ODE (engine)	Open Dynamics Engine
OGRE	Object-Oriented Graphics Rendering
DRL	Deep Reinforcement Learning
EKF	Extended Kalman Filter
IPC	Inter Process Communication

UAV	Unmanned Aerial Vehicle
SITL	Software In The Loop
RK4	4^{th} order Runge-Kutta
BEM	Blade Element Momentum
ECS	Entity Component System
VIO	Visual-Inertial Odometry
CNN	Convolutional Neural Network
RL	Reinforcement Learning
SPOF	Single Point of Failure
PPO	Proximal Policy Optimization
TRPO	Trust Region Policy Optimization
DDPG	Deep Deterministic Policy Gradient
MBRL	Model-Based Reinforcement Learning
MPC	Model Predictive Control
CTBR	Collective Thrust-and-Body Rates
PID	Proportional Integral Derivative
RPM	Rotations Per Minute
SOM	System-On-Module
FPS	Frames Per Second
ToF	Time-of-Flight
DCM	Direction Cosine Matrix
SRT	Single Rotor Thrust
SDF	Simulation Description Format
RTF	Real-Time Factor
OP	Ornstein-Uhlenbeck

SDE	Stochastic Differential Equation
MDP	Markov Decision Process
TD	Temporal Difference
KL	Kullback-Leibler
API	Application Programming Interface
FPV	First Person View
MLP	Multilayer Perceptron
CSI	Camera Serial Interface

1 Introduction

1.1 Motivation

The applications of Unmanned Aerial Vehicles (UAVs) have been growing rapidly over the past few years. The low cost, low power, low footprint, and high maneuverability capabilities of UAVs make them ideal for autonomous applications. Though an under-actuated system, with four actuators and six degrees of freedom, literature has shown us that UAVs can be controlled to stabilize in hovering positions as well as perform maneuvers. UAVs have found their stable positions in a myriad of scenarios across industry, military, and civilian life. UAVs have been used for complex tasks such as cartography, border patrol, search and rescue, forest fire detection, agricultural imaging, and military applications. In a case study conducted by [3], the authors concluded that UAVs serve as a better platform for applications involving agriculture and remote sensing compared to traditional satellites and aircraft.

The environments associated with the majority of real-world UAV applications are complex, which arises from agents, environment, and multi-agent systems that pose heavy communication demand [4]. Conducting experiments in simulation before transitioning to hardware implementation offers a safer, more efficient, and more practical approach [5–7]. Furthermore, simulation tools provide a cheap, fast, and flexible alternative to proto-

type algorithms in multi-UAV scenarios in complex environments. UAV applications are challenging from several perspectives, stemming from the prohibitive policies, the cost of hardware setup and environment setup, as well as the computational requirements. With the emergence and growing significance of machine learning in recent years, UAV applications have expanded in parallel, demanding large datasets capturing system dynamics, which are unfeasible to obtain with real hardware. Simulators play a key role in machine learning applications of UAVs by providing reliable system dynamics to train intelligent agents. [8].

In the context of Perception, Path-planning, and Control applications, the most conclusive and accurate way to validate the algorithms is to perform hardware-based flight testing. However, hardware-based flight experiments to tune controller gains, for instance, can be a safety risk and may result in damage to hardware components [9].

As the real-world applications become more complex, we require model-based approaches to provide solutions. This includes testing and validation of control algorithms before hardware deployment.

Many of the research work around UAVs are conducted in simulations, with a certain degree of uncertainty in the actual hardware implementation [9]. This arises because the results obtained from simulation, such as controller gains and tracking performance, differ greatly in contrast with results obtained from hardware. In technical terms, this is referred to as the Sim-2-real gap. Some of the major concerns around UAV applications are system failures, malicious attacks, and misuse [10]. It is essential to model accurate vehicle dynamics, environment, and communications between UAVs to address these research concerns. Furthermore, many UAV applications are collaborative in nature, which requires precise coordination between agents in a network through communication between agents.

Simulators can also be used as a testbed for prototyping UAV designs, to verify the

feasibility of the design based on aerodynamics before performing flight tests. Performing flight tests with prototyped models can be expensive and time-consuming, as well as dangerous.

In the context of UAV applications, there are three core functions of UAV simulators: Evaluating new technology, Research & Development, and Low-cost training [10]. In particular, the quality of a simulator can be assessed in terms of the following (Fig. 1.1):

- Flight dynamics model
- System model
- Graphics rendering
- Control design
- Sensor modeling
- Physics update
- Ability to support multiple UAV models
- Ability to extend to various environments
- Ability to support deep learning applications

The flight dynamics model represents the dynamics based on the lift-generating mechanism, which is derived based on aerodynamics. This model is dependent on the type of UAV; for instance, fixed-wing UAVs employ an airfoil mechanism, whereas quadrotor UAVs make use of propellers. The system model represents the input-output relationship of the Quadrotor from a control perspective. The system model encompasses the measurement model, the state evolution model, as well as the relationship between control inputs and the state. Graphics rendering is one of the key features that distinguishes many simulators in

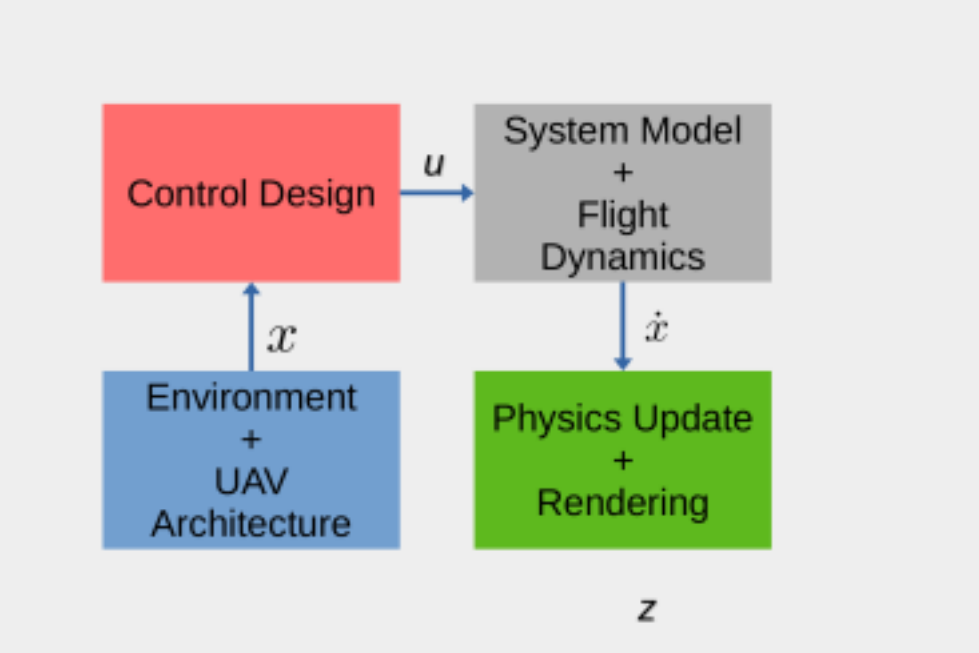

Figure 1.1: The overall orchestration of different modules of a simulator.

the literature and has a significant impact on the Sim-2-real capabilities of the algorithms tested on it. There is a significant trade-off between photorealism and the accuracy of the physics update in reality. Typically, a step size of at least $1ms$ is required for realistic dynamics propagation. However, when photorealistic rendering is also incorporated, the computational load increases substantially with application complexity, often resulting in simulation runtimes that are significantly slower than real-time on most hardware devices. The control design method of a simulator dictates how the user can deploy control systems developed for the System model. Many simulators come with a rigid or fixed control design method, where they only expose the interface for the high-level reference commands while keeping an internal controller fixed. It is necessary to have a flexible control design method, especially in the context of research applications, since the majority of control research is focused on designing control systems that are stable and robust in a variety of

environments.

The current research trends in UAV systems dictate that we have a modular, versatile, scalable, and application-oriented simulator. Furthermore, they shall also support Deep Learning applications, such as Deep Reinforcement Learning (DRL), Vision-based Learning, etc. Recent advancements in Machine learning require huge amounts of annotated data, which can be efficiently collected in simulators [11]. Finally, research applications need to be rapidly prototyped and deployed in hardware, with minimal overhead. In this research, we ask the following research questions:

- How to leverage simulators for application-oriented research and promote reproducibility?
- How to build realistic, scalable UAV simulations with easy control integration?
- How to support deep learning and reinforcement learning experimentation without massive engineering overhead?

To address the **first research question**, we introduce a fully modular simulation platform based on Gazebo sim [12], with interfaces to ROS1 and ROS2. This simulation suite is designed to be utilized in a plug-and-play manner with great flexibility in the capability to extend to diverse applications. The different application scenarios can be modeled using a Simulation Description Format (SDF) file and used with our simulator. Furthermore, our simulator provides baseline environments that are commonly used in UAV research, which can be extended to specific applications. To this end, we also integrate the suite with a digital twin for the Quanser QDrone2. The simulator's modular design enables researchers to utilize individual components (Fig.1.1) in alignment with their specific research objectives. To support control-related studies, a dedicated Control library is provided, allowing the implementation, testing, validation, and simulation of UAV control systems. Finally,

one of the important features of our toolbox is the flexibility to use C++ and Python languages to perform research. This powerful feature allows researchers to conduct fast prototyping of algorithms using Python, and later deploy it using the C++ interface.

To address the **second research question**, we decompose our simulator into a fully modular suite, comprising Planning, Control, and Observation modules. Each of these modules is loosely coupled with ROS1 and ROS2 to ensure that algorithms deployed in the simulator can be transferred to the hardware without further modifications in the source code. The Planning module provides interfaces for high-level trajectory planning by utilizing observations provided by the Observation module. The control module provides the interfaces to deploy custom controllers with our simulation platform, and finally, the Observation module provides the feedback data as well as the sensor data from the environment. The observation module also supports state-estimation techniques such as the Extended Kalman Filter (EKF) based on partial observations of the full state.

To address the **third research question**, we provide an Embodied AI environment, designed based on an OpenAI gym-like environment [13]. One of the key features of gym environments is the ability to drive the environment on the same thread as the training thread, permitting accelerated data collection. High-fidelity simulators often run the environments in a separate process and use Inter-process-Communications (IPC) such as ROS or shared memory to communicate required data, such as Image and IMU data, to the training process. However, such environments are not drivable. High-fidelity refers to the degree to which a simulation, model, or system accurately represents the real-world UAV in terms of dynamics, behavior, and output. A high-fidelity system minimizes simplifying assumptions and incorporates detailed physical modeling, precise numerical methods, and realistic environmental factors, resulting in outputs that closely match empirical or experimental data. In the context of this work, high-fidelity implies that the simulation captures

the critical physical and dynamical characteristics of the quadrotor and its environment to a level that enables reliable performance evaluation and control strategy development. To meet the needs of UAV research with deep learning applications, we provide a Drivable environment as well as an IPC-based environment that can be used to accelerate research in Embodied AI, DRL, etc, based on the PyTorch framework. Furthermore, to take a step further, we provide wrappers to the envpool [14] API, which provides a fast multi-threaded execution of gym-compatible environments. This enables researchers to deploy and test state-of-the-art deep learning algorithms with the simulation suite in an efficient and seamless manner.

In summary, the main objective of this thesis is to develop a modular and application-oriented simulation framework for testing and validating UAV applications across Control and Planning, Computer Vision, and Deep Learning. This framework will serve as a platform to accelerate research for UAVs across Control theory, Computer vision, and Deep Reinforcement Learning.

1.2 Related Work and Challenges

1.2.1 UAV Simulators

UAV control and dynamics simulation are typically performed using standalone multibody dynamics engines or complex software suites such as Gazebo [12], PyBullet [15], and Issac-Sim [16]. Many of the complex suites leverage fundamental multibody dynamics engines such as Bullet [17], MuJoCo [18], Open Dynamics Engine (ODE (engine)) [19], Dart [20], and PhysX [21]. Furthermore, they augment the physics simulation with rendering capabilities, using OGRE (Object-Oriented Graphics Rendering) [22], OptiX [23], Unity [24], and Unreal Engine [25] to provide sensor simulation capabilities. Proprietary tools such

as Matlab & Simulink [26] require a paid license to operate and require manual setup of simulation scenarios, such as multibody dynamics ODE (Ordinary Differential Equations) derivations, constraints, etc. Furthermore, augmentation of rendering is also a paid feature, making extension to complex applications troublesome and expensive.

One of the distinctive features of multibody simulators as opposed to simple ODE solvers is the collision dynamics simulation. Simulators in robotics were originally developed for smooth multi-joint dynamics based on recursive algorithms [27]. Contact simulation, and by extension, collision simulation, are essential for simulating accurate real-world physics in complex environments, where interactions between robots and other agents occur. The approaches to model contact dynamics range from spring-damper models to impulse-based velocity stepping methods. The velocity-stepping method underpins most contact dynamics implementations in fundamental physics engines; however, its performance is constrained by the need to solve NP-hard problems at each simulation step.

Numerical stability of the simulator is an important consideration for simulating complex applications. Many photorealistic simulators often sacrifice physical accuracy to guarantee numerical stability through introducing artificial damping forces or removing Coriolis terms, such as in PhysX [28] and Havok [29]. Furthermore, the mismatch between reality and simulation is governed by two factors in a simulator, the numerical integration error and the system model error [8]. The numerical integration error accumulates due to the local truncation errors, which are a function of h, the timestep, caused by the ODE solvers, and the system model error originates from the uncertainty in the true dynamics variables. Theoretically, as $h \to 0$, the numerical integration becomes exact with real transitions, whereas the system model error can be reduced through proper system identification. The truncation error can be precisely defined as $\epsilon \propto h^n$, where n is the order of the integrator. Most popular physics engines employ 4^{th} order Runge-Kutta (RK4) method, combined

with velocity-stepping contact dynamics simulation.

The majority of the UAV simulators used in research are derived from one of the fundamental engines and augmented with a rendering engine for sensor simulation. Other approaches include employing ad-hoc dynamics propagation using RK4 or other integration schemes at the cost of reduced fidelity. AirSim [11] is a general autonomous vehicles simulator that uses Unreal Engine for rendering and PhysX for dynamics. AirSim presents itself as a simulator for rapid prototyping. FlightMare [30] is a Unity-based flexible quadrotor simulator aimed at Deep Reinforcement Learning based applications. It showcases a detached physics and rendering engine configuration, allowing physics updates up to a rate of 200000 Hz, and a rendering rate of 230 Hz. However, FlightMare suffers from the ad-hoc implementation of physics through an RK4-based scheme. The main motivation to switch to ad-hoc physics is to obtain high-frequency physics; however, this seriously undermines the physical fidelity by ignoring the contact dynamics. RotorS [31] is a classical Gazebo-based general UAV simulator that is tightly coupled to ROS1. It enables researchers to implement control systems for UAVs through a C++ interface. One of the major limitations of RotorS is the outdated rendering capabilities of classic Gazebo (using OGRE 1) and the lack of extendability to Machine Learning applications. Furthermore, integrating the RotorS framework into new applications requires writing software exclusively in C++ and associated build systems, which can be a dead-weight cost in rapid prototyping use cases. Hector [32] is a Gazebo and ROS-based simulator that served as the conceptual backbone for many modern-day UAV simulators. Hector also suffers from the limitations of RotorS.

1.2.2 Simulators Specialized for Control Applications

Extensive testing of control systems is crucial before integrating them into production to ensure safety and performance constraints, particularly for UAV applications. This can be very expensive in terms of time and financial cost when performed with hardware. Prototypes can replace hardware in such scenarios; however, for complex applications and robots, this doesn't provide additional benefits over using hardware. Furthermore, recreating the application-specific environments in a lab can be extremely cumbersome in the initial stages of a project, when there are uncertainties in the feasibility of the application. Real-life implementation of such environments may result in dead-weight cost if the application is unfeasible.

A more reliable approach is to use Simulators with an exact dynamic model to test and validate control systems, and then fine-tune them in the real world. This is motivated by the idea that an accurate model in simulation will provide a rough initial configuration space for the control system parameters, such as gains. However, the quality of the simulated parameter configuration will depend on the accuracy of the dynamic model. For UAV systems, this model needs to be as accurate as possible to reduce the time spent on fine-tuning. The model can range from a simple Newton-Euler formulation, which incorporates translational and rotational dynamics, to more refined models that include Blade Element Momentum (BEM) [33, 34] theory, aerodynamic drag, ground effect, and gyroscopic moments.

The major dynamic effects to account for in a UAV simulation are motor and propeller dynamics, contact dynamics, aerodynamics such as ground effect, and external disturbances, which include wind [32]. Contact dynamics is essential for accurately modeling UAV interactions with obstacles in the environment. Precise contact modeling ensures that behaviors validated in simulation remain reliable when transferred to real-world sce-

narios. Current UAV simulators are polarized, meaning they are specialized for high-quality graphics or pure numerical integration. They trade off important interactions between entities that include collisions, friction, and deformation, for high-speed graphics, as evident in FlightMare and AirSim. The new Gazebo sim [35] offers a perfect harmony between high-fidelity physics and graphics, with high-quality graphics rendering (Fig. 1.2) through OGRE2 or OptiX and highly accurate physics simulation through Dart, Bullet, SimBody [36], or ODE (engine). Furthermore, the ECS (Entity Component System) architecture of the Gazebo sim allows downstream users to extend or modify the simulation execution structure through a C++ API. This flexibility also includes modifying the rendering and physics engine, making it a potent asset for UAV simulators. Notably, Gazebo also showcases high-fidelity contact dynamics, with a customized physics simulation that accounts for friction and deformations of rigid bodies, making it one of the best contenders for UAV simulators.

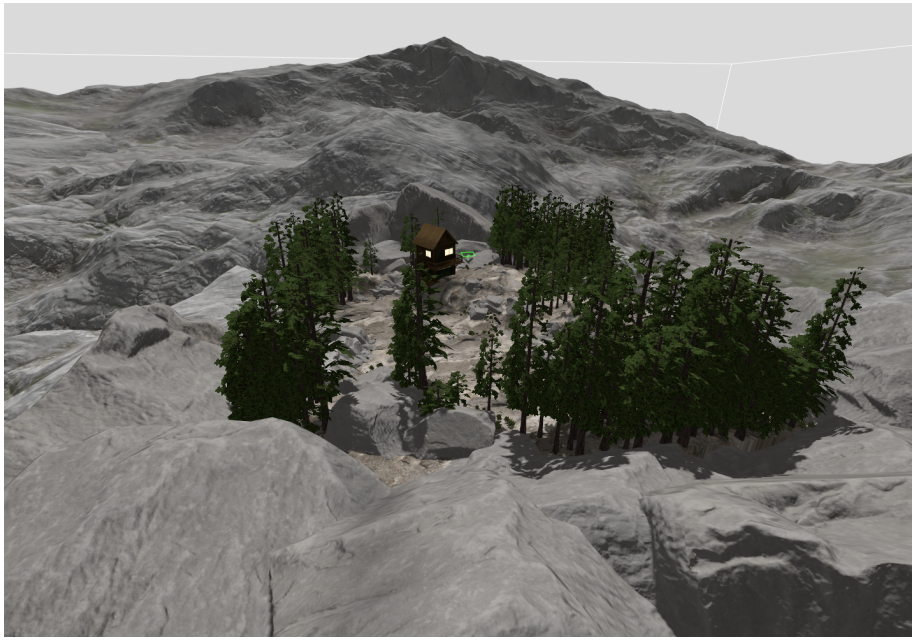

Figure 1.2: Rubicon world rendered in new Gazebo sim

1.2.3 Simulators Specialized for Computer Vision Applications

Computer vision applications mainly revolve around object detection, segmentation, and tracking. Classical computer vision applications in robotics are centered around the state estimation problem, which is realized through Visual-Inertial Odometry (VIO) and Simultaneous Localization and Mapping (SLAM). To conduct vision-based research using simulators is a challenging topic, complicated by issues such as domain shift [37]. Domain shift or domain gap arises from the distribution gap between the data on which the algorithm was fine-tuned and the data it is tested. This is particularly notorious in Machine Learning applications [38] since deep neural networks such as Convolutional Neural Network (CNN) are sensitive to unintended artifacts in the training data, for instance, the texture of the image that will result in a bias.

The problem of transferring an algorithm validated in simulation to real hardware is labeled as the Sim2Real problem [39]. Interestingly, there is an uncertainty when using simulators as a tool for checking the feasibility of a solution, if the simulation model is oversimplified, solving it in simulation may be harder than in real life [39]. Existing techniques for this problem include Domain randomization [40], learning residual dynamics [41] such that $\hat{\mathbf{g}}(.) = g(.) + h(.)$ where $h(.)$ is the residual model and $g(.)$ is the analytical model, and curriculum learning where the algorithm is fit on a small subset of the problem domain, and gradually improved by extending this set.

In terms of UAV applications, computer vision applications include object detection, relative pose estimation with fiducial markers [42], velocity estimation [43], visual odometry [44], and 3D map construction. As noted previously, an ideal simulator should find an optimal tradeoff between high-fidelity physics and rendering, while ensuring adequate compensation for domain shift in high-level applications. The need for physics fidelity is mainly driven by the low-level control applications, and the rendering fidelity is driven by

the high-level applications such as path planning and state estimation. As noted earlier, the flexibility of the Gazebo sim allows users to incorporate different rendering engines, making it a powerful tool to leverage new innovations in rendering engines. Building upon the previous statements, we argue that Gazebo sim offers a reliable tradeoff between graphics and physics, required for complex computer vision-based research that combines high-level and low-level aspects of UAVs. Specifically, the new Gazebo-sim offers more advanced sensor suites required for computer vision applications, including Thermal camera, Multi-camera systems, Segmentation camera, Logical camera, RGBD camera, Depth Camera, and LiDAR, in contrast to other simulators, which makes it a very powerful tool to test and validate any computer vision applications.

1.2.4 Simulators Specialized for Deep Learning Applications

Deep Learning applications of UAV span across the three distinct types of machine learning: Supervised learning, Unsupervised learning, and Reinforcement Learning (RL) [45]. Supervised learning leverages a fixed-size dataset to train a Neural Network to predict labels, whereas unsupervised learning is used for extracting useful information from a dataset. Reinforcement learning differs from the aforementioned approaches and is particularly well-suited for robotics. In most RL algorithms, the simulator contains an environment described by a state $\mathcal{S}$. The agent's objective is to determine the optimal action $\mathcal{A}$ in each state to maximize a scalar reward function $\mathcal{R}$.

The core of mobile robotics is driven by two problems: perception and intelligent control [46]. Machine Learning approaches provide a powerful way to tackle these problems in an end-to-end manner as opposed to traditional and modular approaches, which have several Single points of Failure (SPOF). Furthermore, in the context of perception, traditional methods are reliable only under certain constraints, such as environmental lighting, the

structure of the environment, intensity of the image, etc. Traditionally, perception-related problems relied on handcrafted features to solve, such as object detection, and image recognition [46], whereas, deep learning based approaches can map raw sensory information directly to labels, as in image recognition [47], object detection [48], and semantic segmentation [49].

In the context of intelligent control, DRL offers a promising solution for continuous control through algorithms like Proximal Policy Optimization (PPO) [50], Trust Region Policy Optimization (TRPO) [51], Deep Deterministic Policy Gradient (DDPG) [52], Soft Actor-Critic (SAC) [53]. In [54], the authors demonstrated low-level control of a quadrotor using PPO. The authors of [55] demonstrated the application of Model-Based Reinforcement Learning (MBRL) based Model-predictive Control (MPC) scheme for low-level control of a quadrotor. In [56], the authors demonstrated the first-ever AI drone that outperformed drone racing champions using a learned PPO policy that controls the drone through Collective Thrust-and-Body Rates (CTBR). In [57], the authors demonstrated a dual Policy architecture to train a drone racing policy with a Soft Actor-Critic (SAC) policy for learning a differentiable environment and a PPO policy that learns to solve the racing task across different environments generated by the former.

FlightMare [30], Gym-Pybullet drones [58], AirSim [11] are simulators with DRL training capabilities. FlightMare provides an ad-hoc physics-based environment to provide physics-state only samples up to 200000Hz. However, as noted earlier, the lack of accurate interaction dynamics is one of the major limitations of FlightMare. This has important consequences in DRL when the trained policy is deployed, causing Markov property violations when the robot interacts with the environment, which introduces additional fine-tuning overheads. AirSim also suffers from the simplified multi-body dynamics limitation, and many sensors are only a subset of those available in the new Gazebo-sim, and many are

ad-hoc implemented. In Gym-PyBullet drones, the authors provide a PyBullet-based environment for DRL applications. PyBullet is fundamentally a physics engine and lacks the capabilities for advanced rendering, and similarly to AirSim and Flightmare, the available sensors are only a subset of the Gazebo sensor suite. OmniDrones [59] is a recent DRL environment that allows users to customize the drone models. In [60], the authors presented a physics-only platform that can train RL agents in under 18 seconds.

1.3 Contributions

This thesis makes several contributions to addressing the limitations of existing simulators introduced in Section 1.2, by presenting a new platform suite for UAVS named as GazeboDrones. Specifically, the main contributions of this thesis are as follows:

- A high-fidelity UAV dynamics simulated in Gazebo sim, based on Blade element theory that incorporates disturbances from propeller aerodynamics, ground effect, and environmental disturbances.

- A modular multi-body control-system design platform for UAVs that enables the reuse of existing codebases and integrates new third-party software seamlessly. GazeboDrones provides interfaces for developing, testing, and validating control systems for UAVs that are available through a hybrid C++ and Python library named the GazeboDrones Control library. We demonstrate the applicability of our simulator through the development of control systems for a UAV, starting from a PID controller and progressing to a complex MPC.

- A high-throughput DRL environment to train, test, and deploy deep Neural Network policies that integrate features from the GazeboDrones Control library. Fully compatible with the OpenAI gym API, our environment showcases hybrid operat-

ing modes such as driven and non-driven modes, enabling researchers to simulate realistic scenarios seamlessly. We demonstrate the applications using the Gazebo-Drones DRL library through training an autonomous racing policy, trained across 100 environments in parallel.

- A fully-capable framework for autonomous navigation applications for UAV, including libraries for perception, path-planning, state-estimation, and control of UAV using traditional and deep learning approaches.

- ROS1 & ROS2 support enabling researchers to re-use existing software stacks for real hardware directly with our simulation platform to perform SITL simulation.

1.4 Thesis Organization

This thesis is organized into four chapters, excluding the Introduction. Chapter 2 will provide a comprehensive derivation of the UAV dynamics, with emphasis on Quadrotors. Section 2.1 presents an overview of the Quanser QDrone2 quadrotor, which will be used as our baseline UAV model. Section 3.2 introduces the Gazebo sim software architecture. Section 2.2 introduces preliminary knowledge required for dynamics modeling and control system design for general UAVs. Section 2.3 introduces our high-fidelity control allocation algorithm that maps from virtual controls to actuator control inputs, and subsequently, the simulation of our model.

Chapter 3 discusses our control system toolbox, the backbone of our simulator. Section 3.1 presents an overview of our control system toolbox and the Gazebo simulation API. Section 3.2 introduces the Gazebo sim software architecture. Section 3.3 discusses a new approach to perform simulation through a customized implementation of the Gazebo sim, which enables rapid prototyping of autonomous applications through Gazebo. Section 3.4

presents our control allocation mechanism that leverages the Gazebo sim C++ API, along with a Gaussian measurement error modeling through a Gauss-Markov process. Section 3.5 introduces the control system toolbox of our control library and its features. Section 3.6 introduces the general framework for control UAV control system designs, and simulation and hardware results for several sample controllers. Section 3.7 provides a comparison of our control system toolbox and other SOTA UAV simulators.

Chapter 4 introduces our DRL toolbox built upon our control toolbox and demonstrates the autonomous racing policy training application. Section 4.1 presents an overview of our deep reinforcement learning toolbox. Section 4.2 introduces preliminary knowledge required for training RL policies. Section 4.3 introduces our deep reinforcement learning toolbox and our training platform. Section 4.4 showcases the training process and inference of autonomous racing policy with our toolbox. Section 4.5 provides a performance comparison of samples as the parallelized environments scale.

Finally, the conclusions and future work are presented in Chapter 5.

2 GazeboDrones UAV Modeling

2.1 Overview

This chapter introduces the high-fidelity UAV dynamics modeling and the Control Allocation algorithm implemented in GazeboDRones, the proposed simulation platform. The allocation algorithm is part of a larger framework known as the Control Library, which forms the bedrock of our simulation platform, serving as a pivotal tool for the simulation of numerous UAV applications. It functions as a core unit in the Computer Vision and Deep Learning library. The central motivation for this library is to provide a modular interface to the simulated UAV that can be used for prototyping control systems. As quoted in the introduction, prototyping is an unavoidable step in any control system development. As the application realm of UAVs is expanding rapidly, so is the complexity. Recent advances in UAV technology identified the need for highly precise and accurate modeling, as shown in the champion-level drone racing [56], which pushed the drone hardware and software to its limits.

A simple numerical simulation no longer guarantees performance in highly sophisticated applications that push the drone to its edge. As stated in the introduction section, the new Gazebo sim provides the most fitting features as a simulation platform to meet the growing needs in control system design for UAVs. Besides the accuracy of the multibody simulation,

the precision of the dynamic model is a crucial factor in the reliability of the simulation. To achieve precision, a variety of dynamic effects need to be accounted for: ground effect, downwash in multi-drone simulations, motor dynamics, environmental effects such as wind, atmospheric pressure, and magnetic field.

We utilize the Quanser Qdrone2 Fig. 2.1 as the baseline model for the remainder of this thesis. The drone arrives with NVIDIA System-On-Module (SOM) chip of the Jetson Xavier family, enabling researchers to conduct state-of-the-art UAV research in traditional and machine learning domains.

Figure 2.1: The QDrone2 platform [1].

The platform also boasts a broad array of sensors, including the front-facing Intel RealSense D435 RGBD camera, a downward-facing Omnivision OV989 Grayscale camera that can be operated up to 120 FPS, 3 side-facing Sony IMX219 (Camera Serial Interface) CSI cameras, 2 6-DOF Inertial Measurement Unit (IMU) sensors, 1 ToF sensor, and an optical flow sensor, making it a versatile flying machine suitable for any research applications. The high-performance ARM-based Jetson Xavier GPU, combined with the native support for CUDA, enables reliable real-time performance across an immense amount of computa-

tional load. Applications that demand high computational load and real-time performance include VIO, SLAM, Neural-network controllers, etc.

We begin by presenting a comprehensive dynamic model for the Quanser QDrone2 and finalize with the derivation of the control allocation algorithm. We first detail the full UAV dynamic model, followed by residual dynamics modeling. The dynamic model we use is the Newton-Euler rigid body formulation. This is motivated by the fact that many multirotor and fixed-wing UAV systems are conceivable through a configuration of rigid bodies connected through joints. This configuration comprises a main frame represented as a cube, and four propellers represented as cylinders. For use-cases that require enhanced contact dynamics simulation, the links can be modeled with complex geometries using standard mesh creation software and then exported to be used with our simulation (See Fig. 2.2). However, for the remainder of this thesis, we will assume the former unless explicitly specified.

Figure 2.2: Quadrotor rigid body modeling using SDF.

2.2 Preliminaries

2.2.1 Co-ordinate System

In UAV control system design, there are two primary coordinate systems: the inertial frame and the body frame. The inertial frame is a fixed, non-accelerating coordinate system in which Newton's laws are valid, as opposed to non-inertial (accelerating or rotating) frames that require fictitious forces to describe motion. For the remainder of this thesis, all coordinate frames referred to will be right-handed systems, which satisfy the right-hand rule [61]. The coordinate frames are illustrated in Fig. 2.3, where ${}^{I}(.)$ represents the inertial basis vectors and ${}^{B}(.)$ denotes the body frame basis vectors.

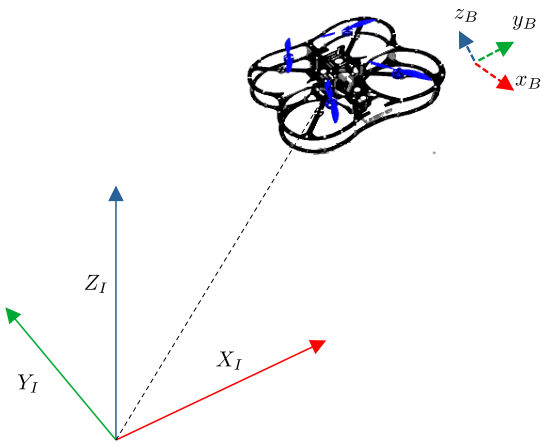

Figure 2.3: The inertial and body frames.

Trajectory control is typically performed in the inertial coordinate system, requiring states to be represented in this frame. However, in attitude control states, such as the angular velocity of the main frame ω_{body}, are represented in the body frame. The orientation can be typically represented as Euler angles $[\phi \quad \theta \quad \psi] \in \mathbb{R}^3$ (Section 2.2.2.1), quaternion $[q_w \quad q_x \quad q_y \quad q_z] \in S^3$ where S^N represents the unit sphere of dimensionality N embedded

in a $N+1$ dimensional space (Section 2.2.2.3), or as rotation matrices $\mathbf{R} \in SO(3)$. $SO(3)$ represents the special orthogonal group, which is the subgroup of the $GL(\mathbb{R}, 3)$, General Linear group that contains the set of all 3×3 invertible matrices with real elements. The orthogonal group is defined as follows,

$$O(\mathbb{R},\ 3) = \{\mathbf{M} \in GL(\mathbb{R},\ 3)\}\ |\mathbf{M}\mathbf{M}^T = \mathbf{M}^T\mathbf{M} = \mathbf{I}\} \tag{2.1}$$

Note that $O(3)$ contains all orthogonal matrices.

$$SO(\mathbb{R},\ 3) = \{\mathbf{R} \in O(\mathbb{R}, 3)\ |\ \det \mathbf{R} = 1\} \tag{2.2}$$

For the remainder of this thesis $SO(3)$ will refer to $SO(\mathbb{R}, 3)$. Furthermore, all inertial frame quantities will be denoted by the superscript ${}^{I}(.)$, and all body frame quantities will be denoted by the superscript ${}^{B}(.)$.

2.2.2 Attitude Kinematics and Dynamics Conventions

The orientation of a rigid body can be represented using Euler angles, quaternions, or Rotation matrices. Several autonomous applications extensively require transformations between coordinate systems, as well as representing physical quantities in intermediate frames to simplify the problem. For instance, in Visual Inertial Odometry (VIO), the feature points are extracted in the camera frame whose orientation can be represented as ${}^{C}_{B}R$, a relative rotation from the body frame to the camera frame. The VIO algorithms can determine the relative rotation matrices ${}^{C}R^{k+1}_{k}$ between images I_k and I_{k+1} using feature correspondences [62] in the camera frame. Given ${}^{C}R^{0} = I$, ${}^{C}R^{k} = R^{0}\prod_{i=1}^{k} {}^{C}R^{i}_{i-1}$. Knowing the fixed rotation ${}^{C}_{B}R$, we can obtain the orientation of the body frame at step k based on the relation ${}^{B}_{I}R{}^{C}_{B}R =^{C}\ R^{k}$

Similarly, in control systems design, physical quantities such as control thrust and moments often need to be transformed to appropriate coordinate systems.

2.2.2.1 Euler Angles

Euler angles are a popular method for attitude parametrization that provides an intuitive method to visualize 3D rotations [63, 64]. This parametrization involves a sequence of rotations with three angles, roll ϕ about the ${}^{I}x$ axis, pitch θ about the ${}^{I}y$ axis, and ψ about the ${}^{I}z$ axis [65]. The resulting rotation matrix can be obtained based on the Direction Cosine Matrix (DCM) [66] The Euler angle convention we use is illustrated in Fig. 2.4. The following notation will be used for simplicity

$$s = \sin,\ c = \cos,\ t = \tan \tag{2.3}$$

The resulting rotation matrix is derived using DCM for individual rotations as follows:

(i) DCM for rotation of roll angle ϕ about ${}^{I}x$, $R_x(\phi)$

$$\begin{bmatrix} {}^{I}x \\ {}^{I}y \\ {}^{I}z \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & c\phi & -s\phi \\ 0 & s\phi & c\phi \end{bmatrix} \begin{bmatrix} {}^{B}x \\ {}^{B}y \\ {}^{B}z \end{bmatrix} \tag{2.4}$$

(ii) DCM for rotation of roll angle θ about ${}^{I}y$, $R_y(\theta)$

$$\begin{bmatrix} {}^{I}x \\ {}^{I}y \\ {}^{I}z \end{bmatrix} = \begin{bmatrix} c\theta & 0 & s\theta \\ 0 & 1 & 0 \\ -s\theta & 0 & c\theta \end{bmatrix} \begin{bmatrix} {}^{B}x \\ {}^{B}y \\ {}^{B}z \end{bmatrix} \tag{2.5}$$

(iii) DCM for rotation of roll angle ψ about ${}^{I}z$, $R_z(\psi)$

$$\begin{bmatrix} {}^{I}x \\ {}^{I}y \\ {}^{I}z \end{bmatrix} = \begin{bmatrix} c\psi & -s\psi & 0 \\ s\psi & c\psi & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} {}^{B}x \\ {}^{B}y \\ {}^{B}z \end{bmatrix} \tag{2.6}$$

The final rotation matrix representing the orientation of the rigid body is obtained by chaining the sequence $(\phi,\ \theta,\ \psi)$ as $R_z(\psi)R_y(\theta)R_x(\phi) = {}^{B}_{I}R$

$$\begin{aligned} {}^{B}_{I}R &= \begin{bmatrix} c\psi & -s\psi & 0 \\ s\psi & c\psi & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} c\theta & 0 & s\theta \\ 0 & 1 & 0 \\ -s\theta & 0 & c\theta \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & c\phi & -s\phi \\ 0 & s\phi & c\phi \end{bmatrix} \\ &= \begin{bmatrix} c\theta c\psi & -c\phi s\psi + s\phi s\theta c\psi & s\phi s\psi + c\phi s\theta c\psi \\ c\theta s\psi & c\phi c\psi + s\phi s\theta s\psi & -s\phi c\psi + c\phi s\theta s\psi \\ -s\theta & s\phi c\theta & c\phi c\theta \end{bmatrix} \end{aligned} \tag{2.7}$$

Note that ${}^{I}V = {}^{B}_{I}R\,{}^{B}V$. Here, ${}^{B}_{I}R$ denotes the rotation matrix from the inertial frame to the body frame, as it transforms inertial-frame unit vectors into body-frame unit vectors. In other words, ${}^{B}_{I}R\mathcal{F}$ rotates frame $\mathcal{F}$, and for the inertial frame $\mathcal{F} := \mathbf{I}^{3\times3}$, this gives ${}^{B}_{I}R\mathcal{F} = {}^{B}_{I}R\mathbf{I}^{3\times3} = {}^{B}_{I}R$. The columns of ${}^{B}_{I}R$ therefore represent the rotated basis vectors. Since the physical quantity itself remains unchanged, its expression in another frame can be obtained by projecting it onto that frame's basis. Consequently, ${}^{B}_{I}R^{T}\,{}^{I}V = {}^{B}V$, and equivalently ${}^{I}V = {}^{B}_{I}R\,{}^{B}V$ (Section 2.2.3). Also note that ${}^{I}_{B}R \in SO(3)$.

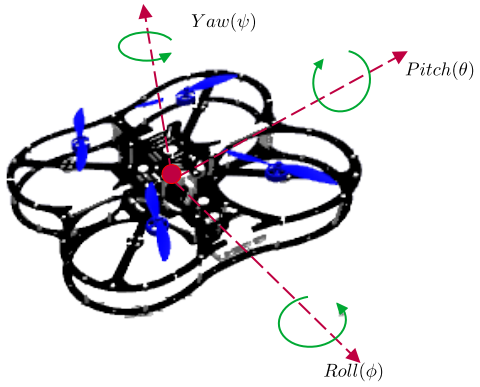

Figure 2.4: The Euler angles convention.

The primary limitation of Euler angles parameterization is that the mapping from $\mathbf{R} \in SO(3) \rightarrow \eta = (\phi,\ \theta,\ \psi) \in \mathbb{R}^3$ has an infinite number of solutions for certain angle sequences; therefore, the true sequence can never be obtained in such configurations.

2.2.2.2 Euler Angles Rates

IMU sensors typically report the angular velocity ω_{body} of the body-fixed frame in the inertial frame as ${}^{I}\omega$ [67], assuming the orientation of the IMU aligns with the body frame. Typically, in UAV dynamics formulations, the attitude terms need to be represented in

the body-fixed frame. To calculate other quantities, it is necessary to transform ${}^{I}\omega$ to ${}^{B}\omega$. For the remainder of this thesis, we will assume all angular velocities and their derivatives are specified in relation to the body-fixed frame. ${}^{B}\omega$ is linearly related to the Euler angle rates $\dot{\eta} = (\dot{\phi},\ \dot{\theta},\ \dot{\psi})$. Instantaneously, angular velocity is a vector and supports vector properties; therefore, we can write it as the sum of Euler angle rates as follows [65]:

$$
{}^{B}\omega = \begin{bmatrix} \dot{\phi} \\ 0 \\ 0 \end{bmatrix} + R_x(\theta) \begin{bmatrix} 0 \\ \dot{\theta} \\ 0 \end{bmatrix} + R_x(\phi)R_y(\theta) \begin{bmatrix} 0 \\ 0 \\ \dot{\psi} \end{bmatrix} = \begin{bmatrix} 1 & 0 & -s\theta \\ 0 & c\phi & s\phi c\theta \\ 0 & -s\phi & c\phi c\theta \end{bmatrix} \begin{bmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix}
$$

$$
{}^{B}\omega = \mathcal{J}^{-1}\dot{\eta}
$$

$$
\mathcal{J} = \begin{bmatrix} 1 & s\phi t\theta & c\phi t\theta \\ 0 & c\phi & -s\phi \\ 0 & s\phi \sec\theta & c\phi \sec\theta \end{bmatrix} \tag{2.8}
$$

The primary issue associated with Euler angle parametrization is that $\mathcal{J}$ is only locally defined [65]. Note that $\mathcal{J}$ is singular when $\theta = \pm\frac{\pi}{2}$, which is one of the major limitations associated with using the Euler angles convention for attitude representation.

2.2.2.3 Unit Quaternions

Unit quaternions have been used for the tracking control of UAVs as demonstrated in [68,69]. They are also particularly useful for Gaussian state estimators, such as the Kalman filter [70], the extended Kalman filter [71], and the unscented Kalman filter [72]. Specifically, they are typically used in the state vector representation of a UAV. Unit quaternions satisfy the following conditions $Q \in S^3$, $Q \in \mathbb{R}^4$, $||Q|| = 1$. For the remainder of this thesis, we will assume a scalar first convention of quaternions where $Q = [q_w\ q_x\ q_y\ q_z]^T$, $\overrightarrow{q}_v = [q_x\ q_y\ q_z]^T$ is the vector part, and the $q_0 = q_w$ is the scalar part. Concisely, we can write $Q = [q_0\ \overrightarrow{q}_v]^T$. The inverse rotation of a quaternion Q is defined as $Q^{-1} = [q_0\ -\overrightarrow{q}_v]^T$,

also known as the complex conjugate. The quaternion product $Q_1 \otimes Q_2$ is defined as

$$\begin{aligned} Q_3 &= Q_1 \otimes Q_2 \\ &= \begin{bmatrix} q0_1 q0_2 - \overrightarrow{q}_{v1}^T \overrightarrow{q}_{v2} \\ -q0_1 \overrightarrow{q}_{v2} + [\overrightarrow{q}_{v1}]_\times \overrightarrow{q}_{v2} \end{bmatrix} \end{aligned} \tag{2.9}$$

here, $[.]_\times$ represents the skew operation or the hat ($\hat{}$) map. Hat map transforms a vector to a skew-symmetric matrix, and the inverse operation is known as the vee ($\vee$) map, where a skew-symmetric matrix is transformed to a vector. For a vector

$$\mathbf{V} = \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix}$$

the $\hat{}$ and $\vee$ maps are as follows:

$$[\mathbf{V}]_\times = \hat{\mathbf{V}} = \begin{bmatrix} 0 & -v_3 & v_2 \\ v_3 & 0 & -v_1 \\ -v_2 & v_1 & 0 \end{bmatrix} \tag{2.10}$$

$$[\mathbf{V}]_\times{}^\vee = \mathbf{V} \tag{2.11}$$

The cross product of two vectors $\mathbf{V_1}$ and $\mathbf{V_2}$ can be represented as matrix multiplication through a hat map transformation of one vector:

$$\mathbf{V_1} \times \mathbf{V_2} = [\mathbf{V_1}]_\times \mathbf{V_2} \tag{2.12}$$

The dynamics of a quaternion are given as

$$\dot{Q} = \frac{1}{2} Q \otimes {}^B\bar{\omega} \tag{2.13}$$

where,

$${}^B\bar{\omega} = \begin{bmatrix} 0 \\ {}^B\omega \end{bmatrix}$$

This can be further expanded and defined as a matrix multiplication as follows:

$$\begin{aligned}\dot{Q} &= \frac{1}{2} Q \otimes {}^{B}\bar{\omega} \\ &= \frac{1}{2} \Psi(Q) {}^{B}\bar{\omega} \\ &= \begin{bmatrix} 0 & -q_x & -q_y & -q_z \\ q_x & q_w & -q_z & q_y \\ q_y & q_z & q_w & -q_x \\ q_z & -q_y & q_x & q_w \end{bmatrix} {}^{B}\bar{\omega}\end{aligned} \tag{2.14}$$

2.2.2.4 Rotation Matrix

The rotation matrix $\mathbf{R} \in SO(3)$ has the benefit of being globally defined and has no singularities. Furthermore, for a given orientation, there exists a single $\mathbf{R}$ as opposed to the multiple possible configurations of Euler angles, and the dual representation of unit quaternions where a single rotation can be represented as $-Q$ or Q. Using quaternions usually requires restriction to a single hemisphere if uniqueness is required [54].

The rotation matrix dynamics are given as follows

$$\dot{\mathbf{R}} = \mathbf{R}[{}^{B}\omega]_{\times} \tag{2.15}$$

2.2.3 Unit Direction Vectors

We define unit vectors of the 3D coordinate system as follows

$$\begin{aligned}\mathbf{e_1} &= \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \\ \mathbf{e_2} &= \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} \\ \mathbf{e_3} &= \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}\end{aligned} \tag{2.16}$$

Finally, for a given rotation matrix R, the unit vectors after rotation is given as

$$\begin{aligned} \mathcal{F} &= \begin{bmatrix} \mathbf{e_1} & \mathbf{e_2} & \mathbf{e_3} \end{bmatrix} \\ {}'\mathcal{F} &= R \cdot \mathcal{F} \\ &= R \cdot \begin{bmatrix} \mathbf{e_1} & \mathbf{e_2} & \mathbf{e_3} \end{bmatrix} \\ &= R \cdot \mathbf{I}^{3\times 3} \\ &= R \end{aligned} \tag{2.17}$$

Here the columns of resulting matrix ${}'\mathcal{F} = \begin{bmatrix} {}'\mathbf{e_1} & {}'\mathbf{e_2} & {}'\mathbf{e_3} \end{bmatrix}$ represents the rotated unit vectors. Transformations of physical quantities are projected onto the new unit direction vectors, $\begin{bmatrix} {}'\mathbf{e_1} & {}'\mathbf{e_2} & {}'\mathbf{e_3} \end{bmatrix}$ for co-ordinate changes. Hence, this can be represented as follows for a vector $V = v_1 \cdot \mathbf{e_1} + v_2 \cdot \mathbf{e_2} + v_3 \cdot \mathbf{e_3}$

$$\begin{aligned} {}'V &= \begin{bmatrix} {}'\mathbf{e_1} \cdot V & {}'\mathbf{e_2} \cdot V & {}'\mathbf{e_3} \cdot V \end{bmatrix}^T \\ &= \begin{bmatrix} {}'\mathbf{e_1}^T V & {}'\mathbf{e_2}^T V & {}'\mathbf{e_3}^T V \end{bmatrix}^T \\ &= \begin{bmatrix} {}'\mathbf{e_1} & {}'\mathbf{e_2} & {}'\mathbf{e_3} \end{bmatrix}^T \cdot V \\ &=' \mathcal{F}^T \cdot V \\ &= R^T \cdot V \end{aligned} \tag{2.18}$$

Similarly, the projection or transformation of a vector ${}'V$ back to original frame $\mathcal{F}$ from ${}'\mathcal{F}$ is

$$\begin{aligned} V &= R \cdot {}'V \\ &= R \cdot R^T \cdot V \quad | \; R \in SO(3) \\ &= I \cdot V \\ &= V \end{aligned} \tag{2.19}$$

2.2.4 UAV State

The state is defined as $x \in \mathbb{R}^{13}$ and represents the complete information required for feedback in control systems designed for UAVs. The state can be specifically represented in different forms and permutations; however, for the remainder of this thesis, we adopt the following convention

$$X = \begin{bmatrix} {}^{\mathbf{I}}\overrightarrow{p} \\ {}^{\mathbf{I}}\overrightarrow{v} \\ \mathbf{Q} \\ {}^{\mathbf{B}}\overrightarrow{\omega} \end{bmatrix} = \begin{bmatrix} {}^{I}x \\ {}^{I}y \\ {}^{I}z \\ {}^{I}v_x \\ {}^{I}v_y \\ {}^{I}v_z \\ q_w \\ q_x \\ q_y \\ q_z \\ {}^{B}\omega_x \\ {}^{B}\omega_y \\ {}^{B}\omega_z \end{bmatrix} \tag{2.20}$$

where, ${}^{\mathbf{I}}\overrightarrow{p} \in \mathbb{R}^3$ is the inertial frame position, ${}^{\mathbf{I}}\overrightarrow{v} \in \mathbb{R}^3$ is the inertial frame velocity, $\mathbf{Q} \in \mathbb{R}^4$ is the orientation represented using quaternion, and ${}^{\mathbf{B}}\overrightarrow{\omega} \in \mathbb{R}^3$ is the angular velocity of the body frame represented in body frame.

2.3 Modeling and Control Allocation

UAVs are underactuated nonlinear systems that are considered classical unstable dynamical systems [73]. To ensure control systems can achieve theoretical stability, we must consider the transformation from virtual control inputs to the lowest-level commands, which are the voltage commands to the motors that modulate the rotor velocities. Control allocation is a mapping from virtual control inputs to the actuator commands while respecting the physical limits, such as saturation, power efficiency, and rates. Control allocation is a

fundamental tool in any control system design that allows users to design control systems without requiring knowledge of actuator-level dynamics [74] since they are dedicated to the allocation algorithm.

In the context of UAVs, and specifically quadrotors, there are 4 parallel rotors configured in a plane, making the system underactuated; 6-DOF with 4 actuators. The virtual control inputs for this system are typically Generalized forces, which consist of the Thrust along the body-fixed z axis and the moments about the body frame, Collective Thrust and Body-rates (CTBR), which consist of the Thrust and the angular rates, or Single rotor thrusts (SRT) [75]

For the remainder of this thesis, we will refer to Generalized forces as the virtual control inputs, unless specified. Under simplified conditions, where aerodynamic effects are neglected, we can construct an Allocation matrix that maps from Generalized forces to the rotor velocities that serve as the control allocation. However, this has significant consequences in aggressive trajectory tracking applications [76]. Nevertheless, this is practically used in the majority of the UAV systems. In real life, unmodeled terms in the allocation algorithm are often the bottleneck between simulation and real flight. To this end, akin to the real QDrone2, we formulate the control allocation algorithm as follows, referring to Fig. 2.5.

2.3.1 Rotor Thrust

First, we consider the rigid body approximation of the QDrone2 defined by the cube with dimensions $L_\phi \times L_\theta \times 0.15$ m^3. The $L_\phi = 0.254$ m is the roll-axis moment arm, which is the distance between $rotor_2$ and $rotor_4$ [1]. Similarly, $L_\theta = 0.2032$ m is the distance between $rotor_1$ and $rotor_2$. The QDrone2 assembly, which includes 4 propellers and motors, a

[1]Note that this is symmetric for $rotor_3$ and $rotor_1$

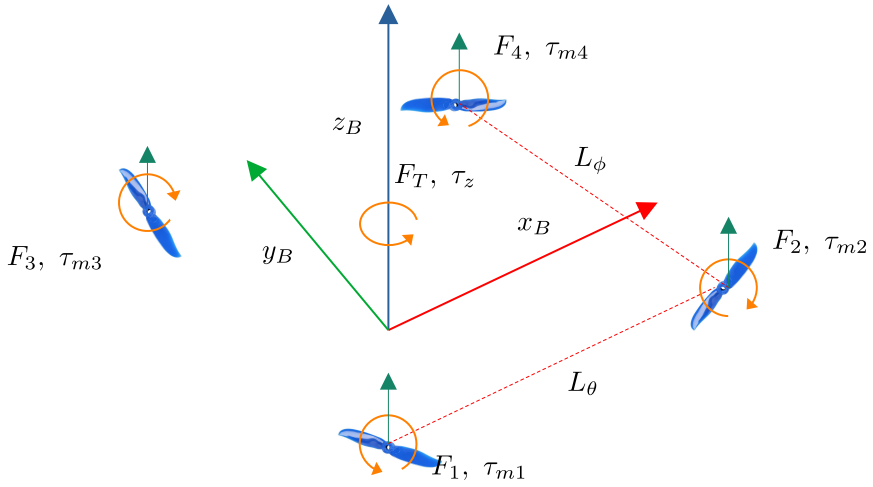

Figure 2.5: The convention used for deriving the allocation matrix.

battery, and the main frame, has a total mass of 1.504 kg.

Thereafter, we define the relationship between the angular velocity of the motor and the generated thrust by the propeller. The motor velocity constant K_V defines the relationship between peak applied voltage V and the output angular velocity ω_{rotor} with no load applied. We consider the experimentally fitted polynomial based on the applied voltage and measured propeller thrust curve to arrive at the following formula for output thrust and the input voltage [1]:

$$T_{rotor}(V) = K_1 V^2 + K_2 V + K_3 \tag{2.21}$$

Utilizing the $K_V = 136.1356$, $K_1 = 0.0362\ \frac{\mathrm{N}}{\mathrm{V}^2}$, $K_2 = 0.117\ \frac{\mathrm{N}}{\mathrm{V}}$, $K_3 = -0.0121$ N and $V = \frac{\omega_{rotor}}{K_V}$ [1], we arrive at the following formulation relating angular velocity to generated

thrust

$$\begin{aligned} T_{rotor}(\omega_{rotor}) &= K_1 V^2 + K_2 V + K_3 \\ &= K_1 (\frac{\omega_{rotor}}{K_V})^2 + K_2 (\frac{\omega_{rotor}}{K_V}) + K_3 \\ &= {}' K_1 \omega_{rotor}^2 + {}' K_2 \omega_{rotor} + {}' K_3 \end{aligned} \tag{2.22}$$

The constants are: ${}' K_1 = 1.9533 \times 10^{-6} \frac{\text{N·s}^2}{\text{rad}^2}$, ${}' K_2 = 0.00085 \frac{\text{N·s}}{\text{rad}}$, and ${}' K_3 = K_3 = -0.0121\text{N}$. Note that ${}^{\mathbf{I}}\overrightarrow{T}_{\mathbf{rotor}} = T_{rotor} {}^{\mathbf{I}}\mathbf{e_3^b}$. Here ${}^{\mathbf{I}}\mathbf{e_3^b}$ is the body frame z axis expressed in inertial frame.

2.3.1.1 Thrust Torques

Referring to Fig. 2.5, the thrust $T_{rotor} = F_i$ generated is not centered at the geometric center of the main frame, which results in roll and pitch moments. Furthermore, due to the rotation of the propellers, a yaw torque is generated which is linearly related to the T_{rotor} by a constant K_τ. In short τ_{m4} in Fig. 2.5 is defined as:

$$\tau_i = \frac{T_{rotor}}{K_\tau} \tag{2.23}$$

Using the definition of moments, we can write

$$\begin{aligned} {}^{\mathbf{B}}\overrightarrow{M} &= \begin{bmatrix} \tau_\phi \\ \tau_\theta \\ 0 \end{bmatrix} + \sum_i \begin{bmatrix} 0 \\ 0 \\ \epsilon_i \tau_i \end{bmatrix} \\ &= \sum_i {}^{\mathbf{B}}\overrightarrow{r_i} \times {}^{\mathbf{B}}\overrightarrow{T}_{\mathbf{i,\ rotor}} + \begin{bmatrix} 0 \\ 0 \\ \epsilon_i \frac{T_{i,\ rotor}}{K_\tau} \end{bmatrix} \end{aligned} \tag{2.24}$$

Here ${}^{\mathbf{B}}\overrightarrow{r_i}$ is the vector to rotor$_i$ from the geometric center of UAV expressed in body frame, and ${}^{\mathbf{B}}\overrightarrow{T}_{\mathbf{i,\ rotor}} = T_{i,rotor} \cdot \mathbf{e_3}$, where $\mathbf{e_3} = {}^{\mathbf{B}}\mathbf{e_3^b} = \begin{bmatrix} 0 & 0 & 1 \end{bmatrix}^T$, and ϵ represents the turning direction of rotor. Note that based on Fig. 2.5 for rotors $(2,3)$ $(\epsilon_2, \epsilon_3) = (-1, -1)$ and for rotors $(1,4)$ $(\epsilon_1, \epsilon_4) = (1, 1)$

Eq. (2.24) can be further expanded based on Fig. 2.5. First we define the position vectors $\overrightarrow{{}^{B}r_i}$ for each rotors:

$$\begin{aligned}
{}^{\mathbf{B}}\overrightarrow{r_1} &= \frac{1}{2}[-L_\theta,\ -L_\phi,\ 0]^T \\
{}^{\mathbf{B}}\overrightarrow{r_2} &= \frac{1}{2}[L_\theta,\ -L_\phi,\ 0]^T \\
{}^{\mathbf{B}}\overrightarrow{r_3} &= \frac{1}{2}[-L_\theta,\ L_\phi,\ 0]^T \\
{}^{\mathbf{B}}\overrightarrow{r_4} &= \frac{1}{2}[L_\theta,\ L_\phi,\ 0]^T
\end{aligned} \tag{2.25}$$

Since ${}^{\mathbf{B}}\overrightarrow{T}_{\mathbf{i,\ rotor}} = \begin{bmatrix} 0 \\ 0 \\ T_{i,rotor} \end{bmatrix}$, the vectors in the cross product become perpendicular. Using the cross-product property, we can write the general form of this product as

$$[x,\ y,\ 0]^T \times [0,\ 0,\ z]^T = [yz,\ -xz,\ 0]^T \tag{2.26}$$

Therefore,

$$\begin{aligned}
\sum_i {}^{\mathbf{B}}\overrightarrow{r_i} \times {}^{\mathbf{B}}\overrightarrow{T}_{\mathbf{i,\ rotor}} &= \sum_i T_{i,\ rotor} \begin{bmatrix} y_i \\ -x_i \\ 0 \end{bmatrix} \\
&= \frac{T_{1,\ rotor}}{2} \begin{bmatrix} -L_\phi \\ L_\theta \\ 0 \end{bmatrix} + \frac{T_{2,\ rotor}}{2} \begin{bmatrix} -L_\phi \\ -L_\theta \\ 0 \end{bmatrix} \\
&+ \frac{T_{3,\ rotor}}{2} \begin{bmatrix} L_\phi \\ L_\theta \\ 0 \end{bmatrix} + \frac{T_{4,\ rotor}}{2} \begin{bmatrix} L_\phi \\ -L_\theta \\ 0 \end{bmatrix}
\end{aligned} \tag{2.27}$$

Concisely, we write

$$\sum_i {}^{\mathbf{B}}\overrightarrow{r_i} \times {}^{\mathbf{B}}\overrightarrow{T}_{\mathbf{i,\ rotor}} = \begin{bmatrix} \frac{-L_\phi}{2} & \frac{-L_\phi}{2} & \frac{L_\phi}{2} & \frac{L_\phi}{2} \\ \frac{L_\theta}{2} & \frac{-L_\theta}{2} & \frac{L_\theta}{2} & \frac{-L_\theta}{2} \\ 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} T_{1,\ rotor} \\ T_{2,\ rotor} \\ T_{3,\ rotor} \\ T_{4,\ rotor} \end{bmatrix} \tag{2.28}$$

Eq. (2.24) becomes

$$
\begin{aligned}
\overrightarrow{{}^{B}M} &= \begin{bmatrix} \frac{-L_\phi}{2} & \frac{-L_\phi}{2} & \frac{L_\phi}{2} & \frac{L_\phi}{2} \\ \frac{L_\theta}{2} & \frac{-L_\theta}{2} & \frac{L_\theta}{2} & \frac{-L_\theta}{2} \\ 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} T_{1,\ rotor} \\ T_{2,\ rotor} \\ T_{3,\ rotor} \\ T_{4,\ rotor} \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ \frac{T_{1,\ rotor}}{K_\tau} - \frac{T_{2,\ rotor}}{K_\tau} - \frac{T_{3,\ rotor}}{K_\tau} + \frac{T_{4,\ rotor}}{K_\tau} \end{bmatrix} \\
&= \begin{bmatrix} \frac{-L_\phi}{2} & \frac{-L_\phi}{2} & \frac{L_\phi}{2} & \frac{L_\phi}{2} \\ \frac{L_\theta}{2} & \frac{-L_\theta}{2} & \frac{L_\theta}{2} & \frac{-L_\theta}{2} \\ \frac{1}{K_\tau} & \frac{-1}{K_\tau} & \frac{-1}{K_\tau} & \frac{1}{K_\tau} \end{bmatrix} \begin{bmatrix} T_{1,\ rotor} \\ T_{2,\ rotor} \\ T_{3,\ rotor} \\ T_{4,\ rotor} \end{bmatrix}
\end{aligned}
\tag{2.29}
$$

2.3.1.2 Linear Thrust Mapping Matrix

Here, we represent the linear thrust mapping matrix for the QDrone2 based on the results obtained from the previous section. Since the total thrust generated along $\mathbf{{}^{I}e_3^b}$, ${}^{B}F_T = \sum_i T_{i,\ rotor}$, we can write

$$
\begin{aligned}
\begin{bmatrix} {}^{B}F_T \\ {}^{B}M \end{bmatrix} = \begin{bmatrix} {}^{B}F_T \\ {}^{B}\tau_\phi \\ {}^{B}\tau_\theta \\ {}^{B}\tau_\psi \end{bmatrix} &= \begin{bmatrix} 1 & 1 & 1 & 1 \\ \frac{-L_\phi}{2} & \frac{-L_\phi}{2} & \frac{L_\phi}{2} & \frac{L_\phi}{2} \\ \frac{L_\theta}{2} & \frac{-L_\theta}{2} & \frac{L_\theta}{2} & \frac{-L_\theta}{2} \\ \frac{1}{K_\tau} & \frac{-1}{K_\tau} & \frac{-1}{K_\tau} & \frac{1}{K_\tau} \end{bmatrix} \begin{bmatrix} T_{1,\ rotor} \\ T_{2,\ rotor} \\ T_{3,\ rotor} \\ T_{4,\ rotor} \end{bmatrix} \\
&= A \begin{bmatrix} T_{1,\ rotor} \\ T_{2,\ rotor} \\ T_{3,\ rotor} \\ T_{4,\ rotor} \end{bmatrix}
\end{aligned}
\tag{2.30}
$$

Finally, we present the linear Motor mapping matrix for the QDrone2 based on the previous result. Based on Eq. (2.30) and (2.22), and making the assumption that lower-

order terms are negligible, we can write

$$\begin{aligned}
\begin{bmatrix} {}^{B}F_T \\ {}^{B}M \end{bmatrix} &= A \begin{bmatrix} T_{1,\ rotor} \\ T_{2,\ rotor} \\ T_{3,\ rotor} \\ T_{4,\ rotor} \end{bmatrix} \\
&= A \begin{bmatrix} {}^{'}K_1\omega^2_{1,\ rotor} \\ {}^{'}K_1\omega^2_{2,\ rotor} \\ {}^{'}K_1\omega^2_{3,\ rotor} \\ {}^{'}K_1\omega^2_{4,\ rotor} \end{bmatrix} = A^{'}K_1 \begin{bmatrix} \omega^2_{1,\ rotor} \\ \omega^2_{2,\ rotor} \\ \omega^2_{3,\ rotor} \\ \omega^2_{4,\ rotor} \end{bmatrix} \\
&= {}^{'}A \begin{bmatrix} \omega^2_{1,\ rotor} \\ \omega^2_{2,\ rotor} \\ \omega^2_{3,\ rotor} \\ \omega^2_{4,\ rotor} \end{bmatrix}
\end{aligned} \tag{2.31}$$

Here, ${}^{'}K_1$ is the rotor thrust constant introduced in Section 2.3.1. The motivation for ignoring lower-order terms is to avoid solving a nonlinear least-squares problem at each iteration of the simulation for solving the matrix equation of the form $A(x \odot x) + Bx + C$, which can be computationally expensive. Furthermore, this is the commonly used allocation matrix form in control system literature.

2.3.2 Disturbance Forces and Moments

Prior to the derivation of Newton-Euler equations, we will report the disturbance terms that are relevant for UAV tracking and control, which are derived using the BEM theory [77]. We will consider the disturbance forces including the drag force ${}^{B}F_D \in \mathbb{R}^3$ due to the interaction of propellers with air during flight, drag moments ${}^{B}M_D \in \mathbb{R}^3$ arising from ${}^{B}F_D$, the aerodynamic drag ${}^{I}F_A \in \mathbb{R}^3$ on the main frame, and the rolling moments ${}^{B}M_R \in \mathbb{R}^3$ [77]. The detailed derivation for these can be found at [77], however, here we report the equations that relate ω_{rotor} and the ${}^{I}v_{rotor}$, the inertial frame velocity (assuming no environmental wind) of the geometric center of propeller to the quantities above (Note that the frame of propellor $\mathbf{P}(.)$ has same orientation as the frame of quadrotor $\mathbf{B}(.)$,

therefore we report the quantities in ${}^{\mathbf{B}}(.)$)

$$
\begin{aligned}
{}^{B}\overrightarrow{F}_{D} &= -\omega_{rotor} C_D {}^{\mathbf{B}}\overrightarrow{v}^{\perp}_{\mathbf{rotor}} \\
{}^{B}\overrightarrow{M}_{\mathbf{R}} &= \omega_{rotor} C_R {}^{\mathbf{B}}\overrightarrow{v}^{\perp}_{\mathbf{rotor}} \\
{}^{B}\overrightarrow{M}_{D} &= -\epsilon C_{M_D} T_{rotor} {}^{\mathbf{B}}\mathbf{e_3^b} \\
{}^{I}\overrightarrow{F}_{A} &= -\mathcal{A}\, {}^{I}\overrightarrow{v}_{\mathbf{Q}}
\end{aligned}
\tag{2.32}
$$

Here, ${}^{\mathbf{B}}\overrightarrow{v}^{\perp}_{\mathbf{rotor}}$ is the projection of the ${}^{\mathbf{I}}\overrightarrow{v}_{\mathbf{rotor}}$ to the plane of rotors expressed in body frame, ϵ denotes the turning direction of each rotor ($+1$ for counter-clockwise and -1 for clockwise), and ${}^{\mathbf{I}}\overrightarrow{v}_{\mathbf{Q}}$ denotes the inertial frame velocity of the main frame. Note that ${}^{\mathbf{I}}\overrightarrow{v}_{\mathbf{Q}}$ is taken to be the relative velocity in the air; however, under no wind conditions, ${}^{\mathbf{I}}\overrightarrow{v}_{\mathbf{air}} = 0$[2]. For a vector $\overrightarrow{u}$, the projection onto the plane of rotors is defined as follows

$$
\overrightarrow{u}^{\perp} = {}^{\mathbf{I}}\mathbf{e_3^b} \times (\overrightarrow{u} \times {}^{\mathbf{I}}\mathbf{e_3^b}) \tag{2.33}
$$

The constant values we use are $C_D = 0.00020673\ \frac{\text{N·rad·m}}{\text{s}^2}$, $C_{M_D} = 10^{-7}\ \frac{\text{N·m}^2\text{·rad}}{\text{s}^2}$, and $C_R = 10^{-8}\ \frac{\text{N·m}}{\text{N}}$. Note that all disturbance forces listed above, with the exception of aerodynamic force, are generated by individual rotors. The matrix $\mathcal{A} \in \mathbb{R}^{3\times 3} = \text{diag}(\begin{bmatrix}0.3 & 0.3 & 0.25\end{bmatrix})\ \frac{\text{N·s}}{\text{m}}$ is the diagonal aerodynamic drag matrix [58]. Note that based on Fig. 2.5 for rotors $(2,3)$ $(\epsilon_2, \epsilon_3) = (-1,-1)$ and for rotors $(1,4)$ $(\epsilon_1, \epsilon_4) = (1,1)$. In addition, we also consider the ground effect, which is a disturbance force relevant when hovering at very low altitudes. The formulation of the ground effect is as follows

$$
{}^{\mathbf{B}}\overrightarrow{F}_{\mathbf{G}} = C_G \omega_{rotor}^2 {}^{\mathbf{B}}\mathbf{e_3^b} \tag{2.34}
$$

where $C_G = 0.0006\ \frac{\text{N·s}^2}{\text{rad}^2}$

[2]Even though we assume no-wind conditions in the derivation, during the simulation we compute the relative velocity w.r.t. air. For readability, we omit the inclusion of ${}^{\mathbf{I}}\overrightarrow{v}_{\mathbf{air}}$

Finally, we also account for the gyroscopic moments that arise due to the rotation of propellers with respect to the rotating main frame. This can be formulated as

$$ {}^{\mathbf{B}}\overrightarrow{M}_{\mathbf{G}} = {}^{\mathbf{B}}\overrightarrow{\omega}_{\mathbf{Q}} \times J_{rotor} \begin{bmatrix} 0 \\ 0 \\ \sum_i \epsilon_i \omega_{i,\ rotor} \end{bmatrix} \tag{2.35} $$

where $J_{rotor} = 4.92 \times 10^{-6}\ \frac{\mathrm{N{\cdot}m{\cdot}s^2}}{\mathrm{rad}}$

In summary, the contribution of forces and moments from a single rotor can be concisely defined as follows

$$ \begin{aligned} {}^{\mathbf{B}}\overrightarrow{F}_{\mathbf{i},\ \mathbf{rotor}} &= \sum F_i \\ &= {}^{\mathbf{B}}\overrightarrow{T}_{\mathbf{i},\ \mathbf{rotor}} + {}^{\mathbf{B}}\overrightarrow{F}_{\mathbf{i},\ \mathbf{D}} + {}^{\mathbf{B}}\overrightarrow{F}_{\mathbf{i},\ \mathbf{G}} \end{aligned} \tag{2.36} $$

$$ \begin{aligned} {}^{\mathbf{B}}\overrightarrow{M}_{\mathbf{rotor}} &= \sum M_i \\ &= {}^{\mathbf{B}}\overrightarrow{M}_{\mathbf{D}} + {}^{\mathbf{B}}\overrightarrow{M}_{\mathbf{R}} + \overrightarrow{{}^{B}r_i} \times {}^{\mathbf{B}}\overrightarrow{T}_{\mathbf{i},\ \mathbf{rotor}} + \begin{bmatrix} 0 \\ 0 \\ \epsilon_i \frac{T_{i,\ rotor}}{K_\tau} \end{bmatrix} - {}^{\mathbf{B}}\overrightarrow{M}_{\mathbf{G}} \end{aligned} \tag{2.37} $$

2.3.3 Newton-Euler Equations

Using Newton-Euler formulation,

$$ \sum_i {}^{\mathbf{I}}\overrightarrow{F}_{\mathbf{i}} = m\,{}^{\mathbf{I}}\overrightarrow{a} \tag{2.38} $$

where $\sum_i {}^{\mathbf{I}}\mathbf{F_i}$ is the total sum of forces expressed in the inertial frame, m is the total mass, ${}^{\mathbf{I}}\mathbf{a}$ is the acceleration of the rigid body (QDrone2), also in the inertial frame, and

$$ \sum_i {}^{\mathbf{B}}\overrightarrow{M}_{\mathbf{i}} = J\,{}^{\mathbf{B}}\dot{\overrightarrow{\omega}}_{\mathbf{Q}} + {}^{\mathbf{B}}\overrightarrow{\omega}_{\mathbf{Q}} \times J\,{}^{B}\overrightarrow{\omega}_{\mathbf{Q}} \tag{2.39} $$

where $\sum_i {}^{\mathbf{B}}\overrightarrow{M}_{\mathbf{i}}$ is the total moments applied in the body frame and $J \in \mathbb{R}^{3\times3}$ is the diagonal inertia (principal moments of inertia) tensor of the entire quadrotor assembly, and ${}^{B}\overrightarrow{\omega}_{\mathbf{Q}}$ is the angular velocity measured in the body frame.

For QDrone2, the inertia matrix (tensor) is defined as

$$J = \begin{bmatrix} J_{xx} & & \\ & J_{yy} & \\ & & J_{zz} \end{bmatrix} = \begin{bmatrix} 0.0177209 & & \\ & 0.0169101 & \\ & & 0.029448 \end{bmatrix} \frac{\mathrm{N \cdot m \cdot s^2}}{rad} \tag{2.40}$$

Substituting Eq. (2.36) to Eq. (2.38) based on the condition that the total sum of individual rotor forces in the inertial frame is the control force

$$\begin{aligned}
\sum {}^{\mathbf{I}}\overrightarrow{F} &= \sum_i {}^{\mathbf{I}}\overrightarrow{F}_{\mathbf{i},\ \mathbf{rotor}} + m\overrightarrow{{}^{I}g} + {}^{\mathbf{I}}\overrightarrow{F}_{\mathbf{A}} \\
&= \sum_i ({}^{\mathbf{I}}\overrightarrow{T}_{\mathbf{i},\ \mathbf{rotor}} + {}^{\mathbf{I}}\overrightarrow{F}_{\mathbf{i},\ \mathbf{D}} + {}^{\mathbf{I}}\overrightarrow{F}_{\mathbf{i},\ \mathbf{G}}) + m\overrightarrow{{}^{I}g} + {}^{\mathbf{I}}\overrightarrow{F}_{\mathbf{A}} \\
&= \sum_i ({}^{B}_{I}R\,{}^{\mathbf{B}}\overrightarrow{T}_{\mathbf{i},\ \mathbf{rotor}} + {}^{B}_{I}R\,{}^{\mathbf{B}}\overrightarrow{F}_{\mathbf{i},\ \mathbf{D}} + {}^{B}_{I}R\,{}^{\mathbf{B}}\overrightarrow{F}_{\mathbf{i},\ \mathbf{G}}) + m\overrightarrow{{}^{I}g} + {}^{\mathbf{I}}\overrightarrow{F}_{\mathbf{A}} \\
&= \underbrace{{}^{B}_{I}R \begin{bmatrix} 0 \\ 0 \\ \sum_i T_{i,\ rotor} \end{bmatrix} + m\overrightarrow{{}^{I}g}}_{nominal\ dynamics} \underbrace{-{}^{B}_{I}RC_D\,{}^{\mathbf{B}}\overrightarrow{v}^{\perp}_{\mathbf{rotor}}(\sum_i \omega_{i,\ rotor}) + {}^{B}_{I}RC_G \begin{bmatrix} 0 \\ 0 \\ \sum_i \omega^2_{i,\ rotor} \end{bmatrix}}_{disturbance\ dynamics} \\
&+ \underbrace{{}^{\mathbf{I}}\overrightarrow{F}_{\mathbf{A}}}_{disturbance\ dynamics} \\
&= m\,{}^{\mathbf{I}}\mathbf{a}
\end{aligned} \tag{2.41}$$

where $\overrightarrow{{}^{I}g}$ is the gravity vector in the inertial frame, and ${}^{B}_{I}R$ is the rotation matrix from the inertial frame to the body frame. Note that the majority of control system designs utilize the nominal dynamics and ignore disturbance terms. This is motivated by the fact that using nominal dynamics reduces the allocation algorithm to a linear thrust mapping matrix. However, during simulation, we ensure all terms in Eq. (2.41) are applied to provide maximum fidelity.

Substituting Eq. (2.37) to Eq. (2.39) we obtain

$$\begin{aligned}
\sum {}^{\mathbf{B}}\overrightarrow{M} &= \sum_i {}^{\mathbf{B}}\overrightarrow{M}_{\mathbf{i},\ \mathbf{D}} + \sum_i {}^{\mathbf{B}}\overrightarrow{M}_{\mathbf{i},\ \mathbf{R}} + \sum_i \overrightarrow{{}^{B}r_i} \times {}^{\mathbf{B}}\overrightarrow{T}_{\mathbf{i},\ \mathbf{rotor}} + \begin{bmatrix} 0 \\ 0 \\ \epsilon_i \frac{T_{i,\ rotor}}{K_\tau} \end{bmatrix} - \sum_i {}^{\mathbf{B}}\overrightarrow{M}_{\mathbf{G}} \\
&= \underbrace{\sum_i \overrightarrow{{}^{B}r_i} \times {}^{\mathbf{B}}\overrightarrow{T}_{\mathbf{i},\ \mathbf{rotor}} + \begin{bmatrix} 0 \\ 0 \\ \epsilon_i \frac{T_{i,\ rotor}}{K_\tau} \end{bmatrix}}_{nominal\ attitude\ dynamics} + \underbrace{C_{M_D}\, {}^{\mathbf{B}}\overrightarrow{v}^{\perp}_{\mathbf{rotor}} \sum_i \omega_{i,\ rotor}}_{} \\
&+ \underbrace{C_R \begin{bmatrix} 0 \\ 0 \\ \sum_i -\epsilon_i T_{i,\ rotor} \end{bmatrix} - J_{rotor}\, {}^{\mathbf{B}}\overrightarrow{\omega}_{\mathbf{Q}} \times \begin{bmatrix} 0 \\ 0 \\ \sum_i \epsilon_i \omega_{i,\ rotor} \end{bmatrix}}_{disturbance\ attitude\ dynamics} \\
&= J^{\mathbf{B}}\dot{\overrightarrow{\omega}}_{\mathbf{Q}} + {}^{\mathbf{B}}\overrightarrow{\omega}_{\mathbf{Q}} \times J^{B}\overrightarrow{\omega}_{\mathbf{Q}}
\end{aligned} \tag{2.42}$$

Similar to the translational dynamics, typical control system designs utilize only the nominal dynamics. However, to ensure high fidelity simulation, we account for all terms in Eq. (2.42).

Important parameters are summarized in Table 2.1.

2.3.4 Motor Dynamics

To push the fidelity of simulation even further, we also include the motor-level dynamics, which are modeled using a first-order low-pass filter. The filter delays the ω_{rotor} with respect to the actual motor velocity reference motor velocity $\omega_{ref,\ rotor}$. This models the exponential rising and falling time, when a motor command is issued. This is represented as follows for the rising ($\omega_{ref,\ rotor} > \omega_{rotor}$) [78]

$$\frac{\Omega_{rotor}(s)}{\Omega_{ref,\ rotor}(s)} = \frac{1}{1 + \tau_{rise} s} \tag{2.43}$$

and as follows for falling ($\omega_{ref,\ rotor} \leq \omega_{rotor}$)

$$\frac{\Omega_{rotor}(s)}{\Omega_{ref,\ rotor}(s)} = \frac{1}{1 + \tau_{fall} s} \tag{2.44}$$

For a discrete time system with sampling time T, this transfer function in the time domain becomes the following [78]

$$\omega_{rotor}(t+T) = e^{\frac{-T}{\tau}}\omega_{rotor}(t) + (1 - e^{\frac{-T}{\tau}})\omega_{ref,\ rotor}(t) \tag{2.45}$$

For QDrone2 simulation we use $\tau_{rise} = \tau_{fall} = 5 \times 10^{-4}$ s.

2.3.5 Allocation Algorithm

Referring to the above-derived equations, our control library manages the disturbance dynamics to simulate high-fidelity quadrotor dynamics, independent of their architecture. After the drone model is defined using the SDF schema, with the appropriate constants defined (see Table 2.1), our control library automatically manages the allocation from virtual inputs to the rotor thrust level (Algorithm 1). The fidelity of the simulation is improved by accommodating the motor dynamics based on equations defined in section 2.3.4. The automatic allocation process involves two parallel simulations: the rigid body simulation by Gazebo and the motor dynamics simulation. The motor dynamics simulation determines a reference rotor velocity $\omega_{ref,\ rotor}$ based on the linear motor allocation matrix defined in Eq. (2.31) and the virtual control inputs $\begin{bmatrix} {}^{B}F_T \\ {}^{B}M \end{bmatrix}$. For the rigid body simulation, the current motor velocity is retrieved to compute the total forces and moments based on Eq. (2.41) and Eq. (2.42). The motor velocity at each rotor joint is then updated based on Eq. (2.45).

To summarize, the allocation algorithm is defined as (see Fig. 2.6)

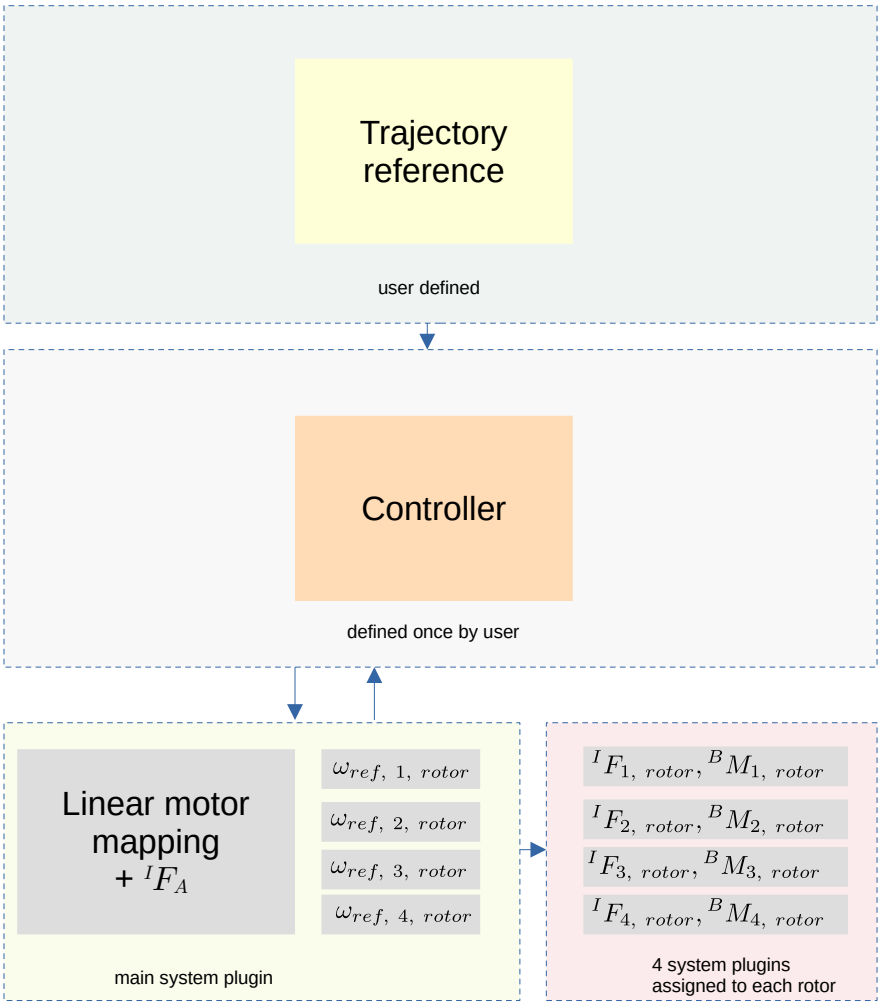

Figure 2.6: The orchestration of our allocation algorithm.

Table 2.1: QDrone2 physical parameters and descriptions

Parameter	**Description**	**Value**
L_ϕ	Roll axis moment arm	0.254 m
L_θ	Pitch axis moment arm	0.2032 m
M_{main}	Mass of the quadrotor frame	1.146 kg
$M_{battery}$	Mass of the onboard battery	0.358 kg
M_{total}	Total quadrotor mass	1.504 kg
K_τ	Motor torque constant	68.9055 $\frac{\mathrm{N}}{\mathrm{N{\cdot}m}}$
K_V	Motor speed constant	136.1356 $\frac{\mathrm{rad}}{V{\cdot}s}$
K_1	Lift coefficient proportional to V^2	0.0362 $\frac{\mathrm{N}}{\mathrm{V}^2}$
K_2	Lift coefficient proportional to V	0.117 $\frac{\mathrm{N}}{\mathrm{V}}$
$K_3 = {}'K_3$	Constant aerodynamic drag force offset	-0.0121 N
${}'K_1$	Lift coefficient proportional to ω_{rotor}^2	1.9533×10^{-6} $\frac{\mathrm{N{\cdot}s^2}}{\mathrm{rad}^2}$
${}'K_2$	Lift coefficient proportional to ω_{rotor}	0.00085 $\frac{\mathrm{N{\cdot}s}}{\mathrm{rad}}$
C_D	Propellor drag coefficient	0.00020673 $\frac{\mathrm{N{\cdot}rad{\cdot}m}}{\mathrm{s}^2}$
C_{M_D}	Propellor drag moment coefficient	10^{-7} $\frac{\mathrm{N{\cdot}m^2{\cdot}rad}}{\mathrm{s}^2}$
C_R	Propellor rolling moment coefficient	10^{-8} $\frac{\mathrm{N{\cdot}m}}{\mathrm{N}}$
C_G	Ground effect coefficient	0.0006 $\frac{\mathrm{N{\cdot}s^2}}{\mathrm{rad}^2}$
$\mathcal{A}$	Aerodynamic drag coefficients	$\mathrm{diag}([0.3,\ 0.3,\ 0.25])$ $\frac{\mathrm{N{\cdot}s}}{\mathrm{m}}$
J_{rotor}	Rotor moment of inertia	4.92×10^{-6} $\frac{\mathrm{N{\cdot}m{\cdot}s^2}}{\mathrm{rad}}$
J	Quadrotor body inertia	$\mathrm{diag}([0.0147209,\ 0.0169101,\ 0.029448])$ $\frac{\mathrm{N{\cdot}m{\cdot}s^2}}{\mathrm{rad}^2}$
$\tau_{rise} = \tau_{fall}$	Motor thrust response time constants	5×10^{-4} s

Algorithm 1: Allocation Algorithm

$N \leftarrow 4$; `// Number of rotors`

for $i \leftarrow 1$ **to** N **do**

$\omega_{ref,\ i,\ rotor} \leftarrow 0$;

$\omega_{i,\ rotor} \leftarrow 0$;

$t \leftarrow$ `simtime()` ; `// Current simulation time`

while *True* **do**

$T \leftarrow$ `simtime()` $- t$; `// sampling time`

$t \leftarrow$ `simtime()` ;

$X_{world} \leftarrow$ `state(world)` ; `// States of all entities`

$X \leftarrow$ `state(quadrotor)` $\in \mathbb{R}^{13}$;

$\dot{X} \leftarrow$ `state_dot(quadrotor)` $\in \mathbb{R}^{13}$;

$X_{ref} \leftarrow$ `reference`$(t) \in \mathbb{R}^{13}$;

$U \leftarrow$ `controller`$(X, \dot{X}, X_{ref}) \in \mathbb{R}^4$;

$\Omega_{ref} \leftarrow \sqrt{{}' A^{-1} \cdot U}$;

$^{\mathbf{I}}\mathbf{v_{air}} \leftarrow X_{world}[\dots]$; `// Velocity of wind`

$R \leftarrow$ `QtoSO3`$(X[7:10])$; `//` $SO(3)$ `from quaternion`

$^{\mathbf{I}}\mathbf{e_3^b} \leftarrow R \cdot \mathbf{e_3}$; `// Body frame z-axis`

$^{\mathbf{I}}\mathbf{F_A} \leftarrow \mathcal{A} \cdot X[4:6]$;

for $i \leftarrow 1$ **to** N **do**

$x_i \leftarrow$ `state`$(rotor_i) \in \mathbb{R}^{13}$;

$\omega_{i,\ rotor} \leftarrow x_i[13]$; `// Rotor angular velocity`

$^{\mathbf{I}}\mathbf{v_{i,\ rotor}} \leftarrow x_i[4:6] - {}^{\mathbf{I}}\mathbf{v_{air}}$;

$^{\mathbf{B}}\mathbf{v}^{\perp}_{\mathbf{i,\ rotor}} \leftarrow R^T \cdot ({}^{\mathbf{I}}\mathbf{e_3^b} \times ({}^{\mathbf{I}}\mathbf{v_{i,\ rotor}} \times {}^{\mathbf{I}}\mathbf{e_3^b}))$;

$^{\mathbf{B}}\mathbf{T_i} \leftarrow ({}' K_1 \cdot \omega^2_{i,\ rotor} + {}' K_2 \cdot \omega_{i,\ rotor} + {}' K_3) \cdot \mathbf{e_3}$;

$^{\mathbf{B}}\mathbf{F_{i,\ D}} \leftarrow -C_D \cdot {}^{\mathbf{B}}\mathbf{v}^{\perp}_{\mathbf{i,\ rotor}} \cdot \omega_{i,\ rotor}$;

$^{\mathbf{B}}\mathbf{F_{i,\ G}} \leftarrow C_G \cdot \omega^2_{i,\ rotor} \cdot \mathbf{e_3}$;

[Continued on next page...]

Algorithm 2: Allocation Algorithm (Part 2 - Continued)

while *True (contd...)* **do**

  for $i \leftarrow 1$ **to** N *(contd..)* **do**

    $^{\mathbf{I}}\mathbf{F}_{\mathbf{i},\ \mathbf{rotor}} \leftarrow R \cdot (^{\mathbf{B}}\mathbf{T}_{\mathbf{i}} + {}^{\mathbf{B}}\mathbf{F}_{\mathbf{i},\ \mathbf{D}} + {}^{\mathbf{B}}\mathbf{F}_{\mathbf{i},\ \mathbf{G}})$; `// Total force generated by rotor`

    $^{\mathbf{B}}\mathbf{M}_{\mathbf{i},\ \mathbf{D}} \leftarrow \epsilon_i \cdot C_{M_D} \cdot {}^{\mathbf{B}}\mathbf{T}_{\mathbf{i}}$;

    $^{\mathbf{B}}\mathbf{M}_{\mathbf{i},\ \mathbf{R}} \leftarrow \omega_{i,\ rotor} \cdot C_R \cdot {}^{\mathbf{B}}\mathbf{v}^{\perp}_{\mathbf{i},\ \mathbf{rotor}}$;

    $^{\mathbf{B}}\mathbf{M}_{\mathbf{i},\ \mathbf{G}} \leftarrow J_{rotor} \cdot X[11:13] \times \omega_{i,\ rotor} \cdot \mathbf{e_3}$;

    $^{\mathbf{B}}\mathbf{M}_{\mathbf{i},\mathbf{rotor}} = {}^{\mathbf{B}}\mathbf{M}_{\mathbf{i},\ \mathbf{D}} + {}^{\mathbf{B}}\mathbf{M}_{\mathbf{i},\ \mathbf{R}} - {}^{\mathbf{B}}\mathbf{M}_{\mathbf{i},\ \mathbf{G}}$; `// Total couple moments generated by rotor`

    `apply`$(^{\mathbf{I}}\mathbf{F}_{\mathbf{i},\ \mathbf{rotor}})$; `// apply at CoG of propellor using Gazebo API`

    `apply`$(^{\mathbf{B}}\mathbf{M}_{\mathbf{i},\mathbf{rotor}})$; `// apply moments at the CoG using Gazebo API`

    $\omega_{ref,\ i,\ rotor} \leftarrow \Omega_{ref}[i]$;

    $\omega_{i,\ rotor} \leftarrow$ `update_filter`$(\omega_{i,\ rotor},\ \omega_{ref,\ i,\ rotor})$

  `apply`$(^{\mathbf{I}}\mathbf{F}_{\mathbf{A}})$; `// apply at CoG of main frame using Gazebo API`

3 GazeboDrones Control System Toolbox

3.1 Overview

This chapter introduces the Control System toolbox for UAVs based on the allocation algorithm developed in the previous chapter. This toolbox is structured as the backbone of our simulation suite, enabling downstream applications such as Reinforcement learning, Vision-based autonomous landing, and path-planning algorithms to be tested and validated in our platform. One of the distinctive features of our Control System toolbox lies in the flexibility to incorporate existing frameworks developed for UAVs into our platform in a language agnostic manner, i.e, C++ or Python. A high-level organization of our toolbox is illustrated in Fig. 3.1

The control system toolbox enables accelerated state-of-the-art research in the UAV autonomous applications through the high-fidelity dynamics modeling and control allocation derived in Section 2.3.5, and the perfect ecosystem for real-world deployment provided by Gazebo sim [59]. We realize the allocation algorithm by leveraging the Gazebo sim C++ API [79], which exposes all parts of the simulation flow, enabling downstream users to integrate hooks and modify simulation behavior programatically. Specifically, we achieve the functionality illustrated in Fig. 3.1 through a set of system plugins. Furthermore, our toolbox comes with native ROS1 and ROS2 support that enables researchers to integrate

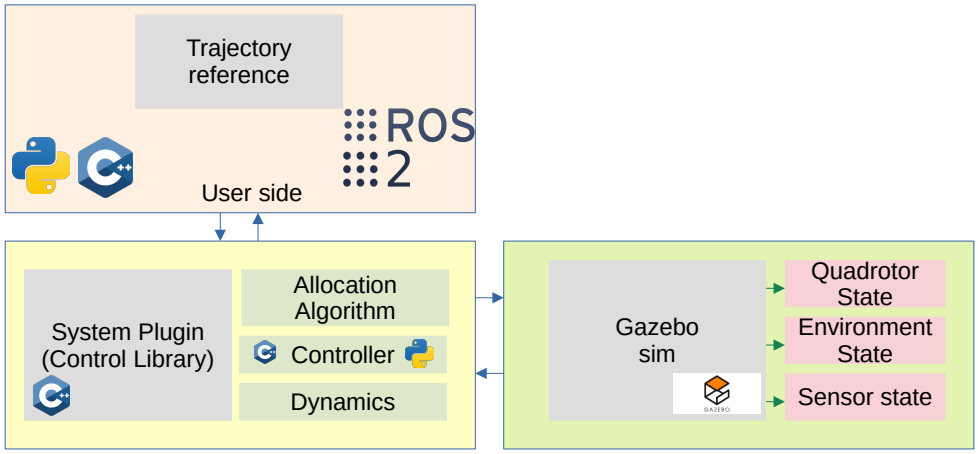

Figure 3.1: An illustration of the high-level organization of the Control library.

community-contributed ROS packages directly into the design and testing of control systems. We provide a ROS-oriented eco-system for researchers who prefer to completely work under ROS middleware, which includes features to record experiments as `rosbags` and validate them in an offline manner, enabling high sample efficiency in resource-constrained systems as well as data sharing across the research community.

3.2 Gazebo Sim Architecture

We begin by first presenting a comprehensive overview of the Gazebo sim architecture, before diving into our toolbox. The Gazebo sim can be generally referred to as a collection of Gazebo libraries whose main entry-point is the Gazebo simulation [2]. The suite of libraries include the `gz-common`, `gz-msgs`, `gz-transport`, `gz-sensors`, `gz-sim` etc. Each library provides a unique functionality to the entire simulation, while providing flexibility to programatically modify its behavior.

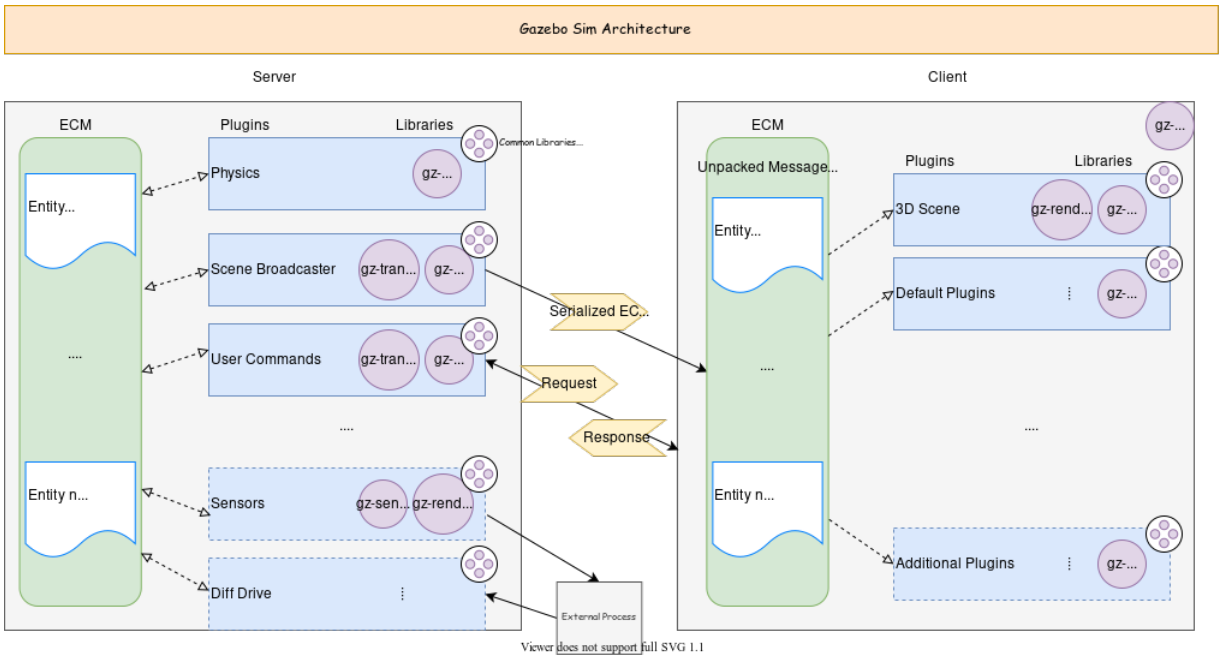

Figure 3.2: The Gazebo sim architecture [2].

This is enabled through the Entity Component System (ECS) architecture (Fig. 3.2) which follows a pattern where each object in the simulation is abstracted as an Entity, whose specific behavior is determined by the components it has. Standard Gazebo API is packaged with built-in Components such as sensors, actuators, collision, pose, etc; however, the API also permits the addition of custom components. Entities can be built by defining a baseline model using the SDF schema, and operated upon during simulation by mutating their components. The baseline SDF model defines a set of essential components such as the pose information, sensor data, collision information, actuator information, etc. This powerful design permits downstream users to add, inspect, modify, or remove Components during the simulation pipeline, making Gazebo a very powerful simulation tool and an invaluable asset for prototyping UAV applications.

Specifically, our control system toolbox leverages this ECS feature through a set of System plugins.

System plugins are Interface classes defined by the Gazebo C++ API. This set include five pure interface classes; `ISystemConfigure`, `ISystemPreUpdate`, `ISystemUpdate`, `ISystemPostUpdate` and `ISystemReset`. Customized system plugins leverage the multiple inheritance feature of C++ to inherit from a subset of the base system interfaces and implement the functionality of each inherited interface. For instance, inheriting `ISystemConfigure` requires the child class to implement the `Configure` method, which will be called once during the simulation, when the entity to which it is attached is initialized. In a similar manner, `ISystemPreUpdate` implements the `PreUpdate` method, which allows the plugin access to the Entity Component Manager (ECM) object, which holds a global directory of all entities in the simulation. Through ECM, the system plugin can access any desired entity, quadrotor for instance, and proceed to inspect, modify, add, or remove components associated with it.

At each simulation step, the Gazebo sim undergoes three phases: the pre-update, update, and post-update phases. Pre-update phase prepares all entities in the simulation to be updated by the physics and rendering engine. The simulation running algorithm of Gazebo sim invokes the `PreUpdate` method of all system plugins that inherit from `ISystemPreUpdate`. The update phase performs the state update based on the Gazebo physics engine and updates the scene based on the new physics state. The updated scene is then utilized by sensors to generate the sensor data. Furthermore, the simulation runner invokes the `Update` method of all system plugins that inherit from `ISystemUpdate`. Finally, in the post-update phase, all entities become immutable. This phase primarily exists to perform cleanup for resources allocated during the pre-update or update phase. However, this can also be utilized to obtain the state information of any entity in the system, to provide feedback, for instance. Note that during the pre-update and update phase, the ECM object permits mutation of entities, but not in the post-update phase.

The system plugins are entirely written in C++, leveraging the Gazebo API. After the definition of the plugins, they can be compiled using a build ecosystem to produce a shared library object. This can then be utilized in the simulation by defining the system plugin in the SDF, and specifically attaching it to some entity.

3.3 Steppable Simulation and Customized Server

The default Gazebo sim software has two parallel parts, a GUI and a Server. The Server is responsible for the management of entities in the simulation, and ensuring the simulation running algorithm is executed in the proper order as described in Section 3.2, whereas the GUI provides user-side utilities such as visualization, interaction with the simulation objects, etc. In the majority of use-cases of Gazebo, the default setup is employed, where a server and GUI run in parallel, and the users leverage inter-process communication (IPC) to interact with entities in simulation (Fig. 3.3). Note that in this setup, the user has no control over the stepping process of simulation, i.e, the simulation cannot be stepped on-demand. In the default configuration of Gazebo software, the users can specify a Real-Time Factor (RTF) and physics step-size in the SDF file, which is utilized by the Server to determine the number of steps taken per second. The relationship between steps and the RTF is

$$\frac{steps}{sec} = \frac{RTF}{T_{physics}} \tag{3.1}$$

For applications that require HITL validation, RTF is set to 1, which implies that the simulation is one-to-one with real life. However, the RTF specification is not always guaranteed to be constant across the simulation duration since the computational time for a single simulation step will vary depending on the complexity of the simulation. Furthermore, the default configuration has serious implications in certain applications that rely

on IPC to interact with the Gazebo sim. For instance, an algorithm running in a separate process, such as a VIO algorithm to estimate pose from a sequence of images, uses its own wall-clock time to track the time interval between inputs coming from the simulation, such as the rendered image, and ground truth data. If the RTF specification is not guaranteed, then the assumptions undertaken at the user-side algorithm will be broken, hence, invalidating the results from the algorithm.

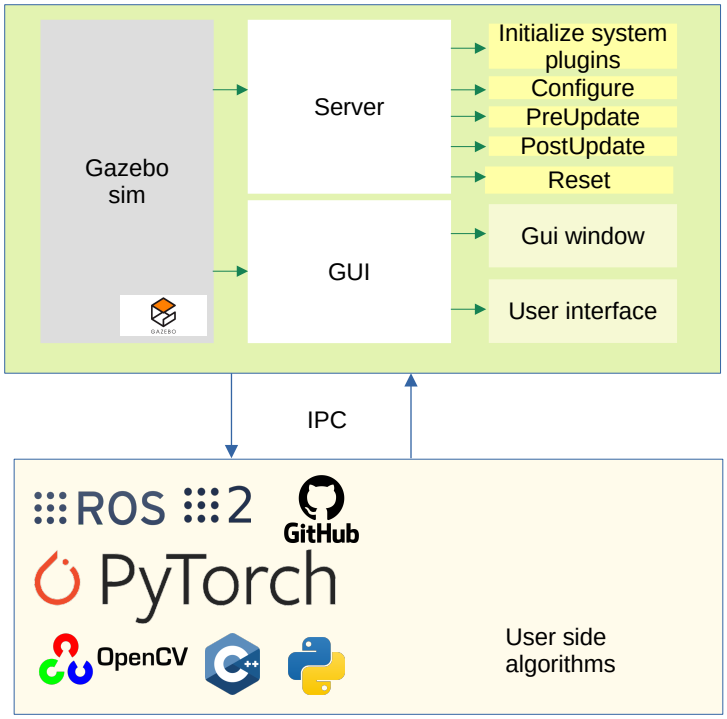

Figure 3.3: The organization of Gz software in the default configuration.

Another limitation of the default configuration of Gazebo is that experiments conducted on one device can almost never be reproduced on another device, and could potentially not even be reproducible in the same device. Reproduction of research work is essential to ensure the credibility of the results. Furthermore, in certain Reinforcement learning appli-

cations, the training process typically involves collecting exploration trajectories based on initial, junction, and branch trajectories [54]. This approach collects a set of initial trajectories using the current policy (on-policy) and generates junction trajectories by adding additive Gaussian noise to the initial trajectory, from which the branch trajectories are generated. This type of data collection assumes that the temporal spacing of trajectories is uniform and that multiple junctions can be created from a given end-state from initial trajectory, which is impossible with the default configuration of Gazebo, since there is always a non-deterministic delay when interacting with the entities in simulation using IPC, the branching trajectories can never be properly collected. Finally, recreating the end-state from the initial trajectory is almost never possible with the IPC-based simulation due to its non-deterministic nature.

To overcome the aforementioned limitations, our simulation suite provides a new approach to running the Gazebo simulation that completely detaches the GUI and Server. We name it the drivable environment, which allows the Server to be created by the downstream user and synchronously pause and step the Gazebo simulation on demand. This guarantees synchronization between the user-side algorithm and the simulation side, since both are executed in the same process with no overhead from IPC as well as the flexibility to drive the simulation on-demand. Specifically, we introduce a `CustomServer` (Fig. 3.4) that abstracts the inner workings of the Gazebo server, while managing the interface between the user-side and simulation side. This abstraction forms the backbone of our Control, Computer vision, and Reinforcement learning toolboxes of our simulation suite, and greatly extends the capabilities of Gazebo as a research tool in robotics.

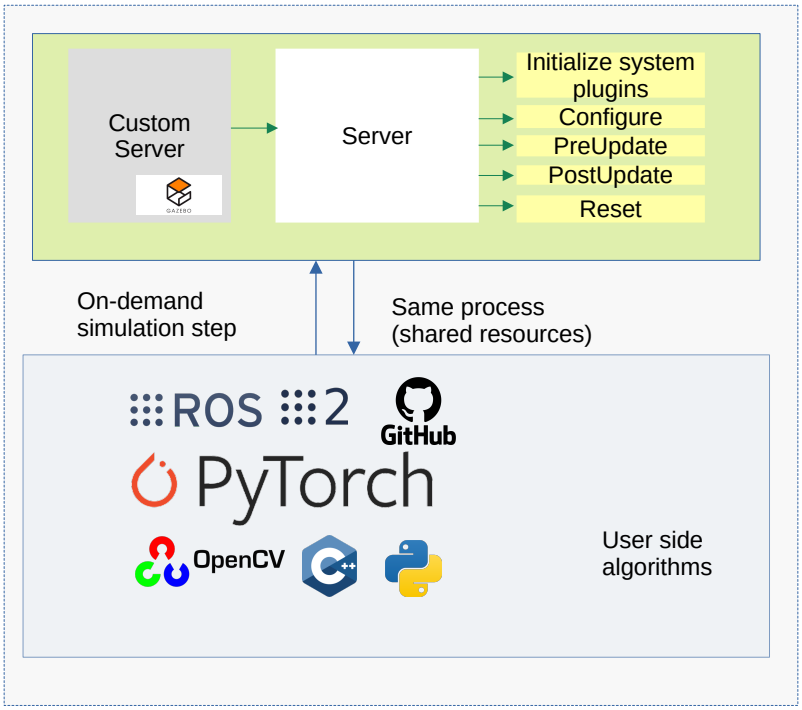

Figure 3.4: The organization of our `CustomServer`.

Furthermore, we also provide the flexibility to create custom servers in Python through Python extension module bindings, greatly diversifying the potential use-cases.

3.4 Allocation System Plugins

To facilitate the high-fidelity simulation of UAVs utilizing the control allocation algorithm derived in Section 2.3.5, we develop two system plugins as part of our control library. The first plugin, named `ServerInterface`, is responsible for managing the interface between the user-side algorithm and the UAV in Gazebo-sim. The managed algorithms, for instance, a controller, can be written in C++ or Python. This is a global plugin that is attached

to the World entity in Gazebo sim by our `CustomServer` (Section 3.3). This binding allows our plugin to access, read, and modify components of every entity in the simulation. Specifically, we can obtain the 6 DOF pose, linear and angular velocity, linear and angular acceleration, collision data, and sensor data, as well as apply control inputs such as torques and forces for any desired entity. The motivation for having this plugin is to extract the feedback information from the UAV as well as the apply disturbance terms that directly act on the main frame of the UAV and are not generated by the propeller dynamics, such as the aerodynamic drag ${}^{\mathbf{I}}F_A$. Furthermore, it provides a convenient way to manage the individual rotors by first mapping virtual inputs to rotor velocities using the linear motor mapping matrix, followed by the distribution of rotor commands to appropriate rotors.

Next, to further enhance the fidelity of simulation, we develop the `RotorPlugin`, a system plugin that manages the rotor-level dynamics and applies control input allocation at the rotor level. This is a shared library plugin that is attached to each rotor and is compiled to a shared object. This plugin is attached to each rotor by specifying it in the SDF file. Specifically, each rotor in the UAV, quadrotor in our case, gets a dedicated `RotorPlugin`. This design approach ensures that the system plugin is attached to the proper entity and the user has the flexibility to define arbitrary UAVs with any number of rotors through the SDF file. Similar to `ServerInterface`, `RotorPlugin` can also access, read, and modify components of every entity in the simulation. However, we restrict this plugin to only access the rotor link and the rotor joint entities to keep our library design modular. During the simulation, each rotor will receive a reference command $\omega_{i,\ ref,\ rotor}$ from the `ServerInterface`, and then perform the allocation logic derived in Section 2.3.5.

During the creation of the `CustomServer`, we first create a traditional Gazebo Server and load the SDF file specified by the user. Followed by initialization of the world and all entities by Gazebo Server, which includes the UAV and the dedicated `RotorPlugin`s, we

attach the `ServerInterface` plugin to the Server using our `CustomServer` class. At any phase of the simulation, the `CustomServer` class can inspect and modify any object in the simulation through a shared pointer to `ServerInterface` plugin. This forms the backbone of our versatility feature, enabling diverse algorithms to be used in conjunction with our toolbox (see Fig. 3.5).

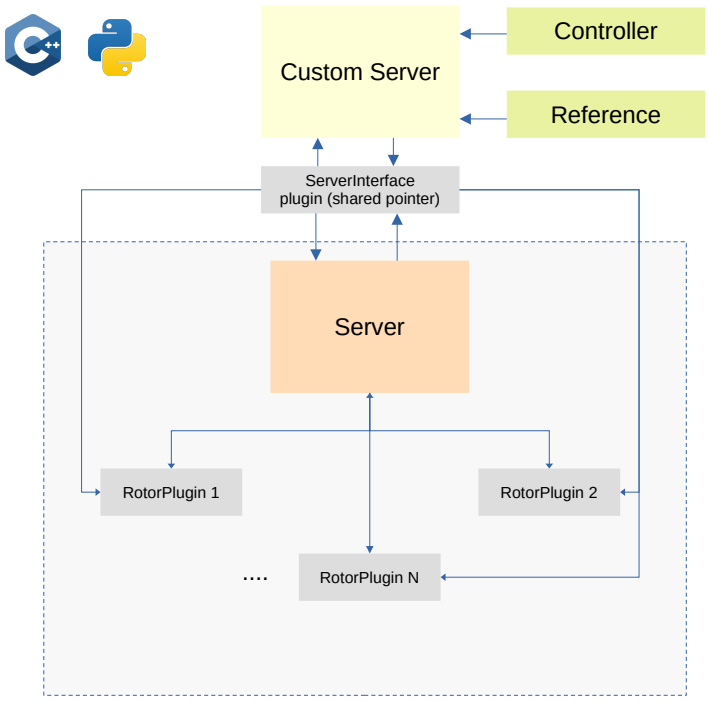

Figure 3.5: The organization of System plugins provided with our control library, along with the custom server

In addition to the high-fidelity dynamics modeling derived in Section 2.3.5, we also consider additive noise dynamics, which are Gaussian and Markov in nature. Specifically, we consider a class of processes named Gauss-Markov processes that satisfy the Markov property as well as the Gaussian model. This is motivated by the fact that every known

dynamic model is incomplete; that is, there cannot be a perfect model. The dynamics can then be reformulated as follows

$$\dot{X} = f(X, u) + g(X, u) \tag{3.2}$$

where $f(X, u)$ is the known dynamics, and $g(X, u)$ is the unknown dynamics. Specifically, we follow the assumption that $g(X, u)$ is Gaussian in nature [80]. In particular, we use the Ornstein-Uhlenbeck (OP) process [81]. Unlike Gaussian white noise, the Ornstein-Uhlenbeck process introduces time-correlated noise that models a slowly varying stochastic influence. This type of modeling can account for ignored terms in aerodynamic effects, actuator-level dynamics, and even environmental disturbances. This modeling choice provides a more realistic model for testing controllers, bridging the gap between simulations and real-world implementation. Finally, this type of noise modeling is also used in deep reinforcement learning applications to simulate uncertainty, as shown in [82].

Note that by using the Ornstein-Uhlenbeck process, we enforce that the noise is spatially uncorrelated, i.e, the covariance is 0 between each physical quantity, but temporally correlated.

The OP can be represented as the following Stochastic Differential Equation (SDE) [83], for a Y defined as the position of a particle

$$\ddot{Y} = -\theta\dot{Y} + \sigma\xi \tag{3.3}$$

Here, θ is the friction coefficient, σ is the diffusion coefficient, and ξ is white noise, usually simulated by a Wiener process. In particular, the associated velocity process can be considered as a Langevin equation, and defined as

$$d\dot{Y} = -\theta\ \dot{Y}dt + \sigma\ dW \tag{3.4}$$

In a discrete-time simulation, we can write this based on the Euler-Maruyama method

as

$$\begin{aligned}\dot{Y}_{k+1} &= \dot{Y}_k - \theta\dot{Y}_k + \sigma\Delta W_n \\ &= {}^{'}X_k - \theta^{'}X_k + \sigma\Delta W_n\end{aligned} \tag{3.5}$$

Here, ΔW_n is independent and identically distributed (i.i.d) Wiener increments of Gaussian noise $\mathcal{N}(\mathbf{0}, \mathbf{\Delta t})$. Precisely,

$$\Delta W_n \sim \mathcal{N}(\mathbf{0}, \mathbf{\Delta t}) = \sqrt{\Delta t}\, \mathcal{N}(\mathbf{0}, \mathbf{1}) \tag{3.6}$$

In the context of our simulation suite, we corrupt our state vector X with OP as follows

$$X_{OP}(t) = X(t) + {}^{'}X(t) \tag{3.7}$$

with

$$\mathcal{B}_N(a, b) = \mathbf{I}^{N\times N} \cdot a + \mathbf{1}^{N\times N} \cdot b$$

$$\theta = \begin{bmatrix} \mathcal{B}_3(0.01,\ 0.03) & & & \\ & \mathcal{B}_4(0.01,\ 0.03) & & \\ & & \mathcal{B}_3(0.01,\ 0.03) & \\ & & & \mathcal{B}_3(0.01,\ 0.03) \end{bmatrix} \tag{3.8}$$

$$\sigma = \begin{bmatrix} \mathcal{B}_3(0.005,\ 0.003) & & & \\ & \mathcal{B}_4(0.005,\ 0.003) & & \\ & & \mathcal{B}_3(0.005,\ 0.003) & \\ & & & \mathcal{B}_3(0.005,\ 0.003) \end{bmatrix} \tag{3.9}$$

, and

$$\Delta W_n = \sqrt{\Delta t}\, \mathcal{N}(\mathbf{0}^{13}, \mathbf{1}^{13\times 13}) \tag{3.10}$$

where $\mathbf{1}^{N\times N} = \begin{bmatrix} 1 & \dots \\ \vdots & \ddots \end{bmatrix}$

Specifically, the measured states provided by our `ServerInterface` plugin (and by extension the `CustomServer`) will be X_{OP}. Furthermore, we also perform the same procedure with the derivatives of the state.

3.5 Control Library

Building upon the drivable environment and our system plugins, we introduce the Control System toolbox. This toolbox serves as a critical tool for researchers in the UAV control theory field to test and validate control schemes in a safe, reliable, and fast manner, leveraging the ecosystem provided by Gazebo sim and the high-fidelity modeling suitable for HIL simulations derived in Section 2.3.5. The main contributions of this toolbox are as follows

- A modular platform for design and validation of control systems for arbitrarily defined UAVs, with precision dynamic modeling (Section 2.3.5) derived from Blade element theory combined with additive dynamics (Section 3.4) simulated in a Gazebo-based eco-system
- ROS1 & 2 support enabling quick simulation to hardware transfer
- Full support for C++ and Python, enabling researchers to rapidly prototype applications in user-friendly Python and test performance-critical algorithms in the optimized C++
- Full open-source software support that allows researchers to directly deploy community-contributed packages into algorithm design
- Support for UAV control at various Virtual control inputs, ranging from Generalized forces and rotor thrusts to rotor velocities.

3.5.1 Control System Design

To facilitate rapid prototyping of control algorithms, as well as abstract the complicated Gazebo C++ API and the control allocation from the user-side, we define interface classes

in C++ and Python for Controllers, named `ControllerBase`. This design approach ensures any controller that inherits from `ControllerBase` is fully compatible with our library. Furthermore, this allows the definition of arbitrary control laws encapsulated within the source code of the customized control law. Inheriting `ControllerBase` enforces the child class to implement the virtual methods (in Python this is referred to as an abstract method) `calculate_u` and/or `calculate_actuator_cmd`. The former method implements the virtual control law, and the latter implements the actuator level control law, which is then executed at a desired control frequency by the `CustomServer` and appropriately processed. Note that implementation of `calculate_actuator_cmd` is optional, and the user can explicitly specify to utilize actuator-level control instead of virtual control. Control frequency is the rate at which the controller runs in real-time as a baseline, which shall satisfy the following constraint

$$\begin{aligned} f_{control} &\leq f_{physics} \\ f_{physics} &= \frac{1}{T_{physics}} \\ \frac{steps}{control} &= \frac{f_{physics}}{f_{control}} \end{aligned} \tag{3.11}$$

For a step size of 1 ms, $f_{physics} = 1000$ Hz, and $f_{control} = 100$ Hz, we have $\frac{steps}{control} = 10$. In other words, the simulation steps 10 times for each control input. This feature allows researchers to test control algorithms at different frequencies, enabling validation of algorithms that need to be run in resource-constrained systems. Furthermore, this is also useful in reinforcement learning algorithms, where the policy is usually executed at very low frequencies.

3.5.2 `calculate_u`

The object-oriented design of `ControllerBase` allows retention of arbitrary data as member variables. Combining this feature with the feedback information and the reference

states provided by our `CustomServer`, users can implement arbitrary control laws within the function body of `calulate_u`. Specifically, `calculate_u` takes as arguments the state vector of the UAV X, its derivative $\dot{X}$, and the reference states X_{ref} as inputs. The X and $\dot{X}$ are provided through our `CustomServer`, and X_{ref} is defined by the user at run time. The output of this method is the virtual control input, or Generalized Forces. The virtual control inputs are then allocated to each rotor based on the logic defined in previous sections.

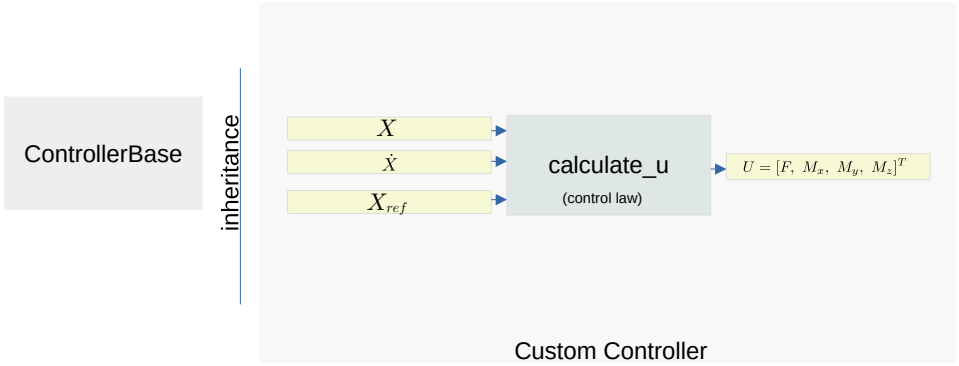

Figure 3.6: calculate_u method

3.5.3 `calculate_actuator_cmd`

Following the design pattern for `calculate_u`, `calculate_actuator_cmd` provides more flexibility to researchers, who prefer to work at the lowest control level. The key difference from `calculate_u` is that the control inputs are now $\omega_{ref,\ i,\ rotor}$, the reference angular velocities of each rotor (Fig. 3.7). Furthermore, the user needs to explicitly specify that the desired controller type is actuator level, which is done during the initialization of the `CustomServer`. Finally, when using this approach, the `ServerInterface` side pre-allocation processing is not performed; rather, the control commands directly go to the `RotorPlugin`s, which also manage the first-order rate limits on the control commands.

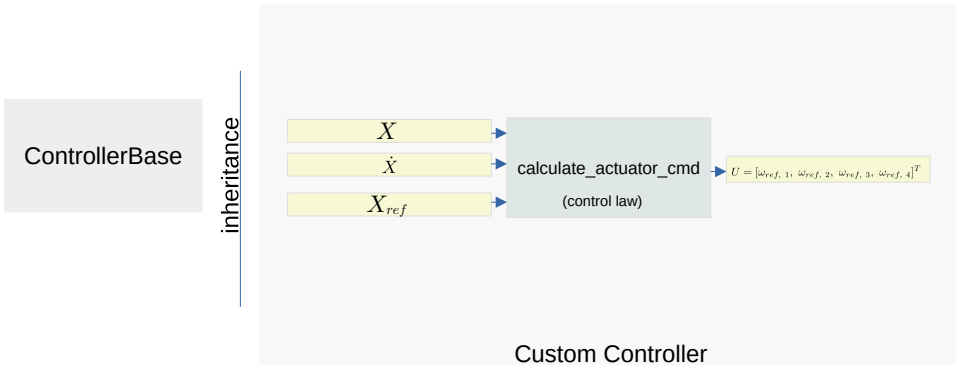

Figure 3.7: calculate_actuator_cmd method

3.6 Applications

Leveraging our control system toolbox for UAVs, we now demonstrate several applications first in simulation, followed by experimental validation in the actual QDrone2. First, we demonstrate the implementation, testing, and validation of a PID controller, followed by a Geometric Controller in $SE(3)$, and a $2D$ NMPC for trajectory tracking with slung payload using an open-source OCP solver.

3.6.1 Simulation-Hardware Applications

3.6.1.1 PID Controller

The Proportional-Integral-Derivative (PID) controller is the most common yet effective control algorithm in automatic control. An enormous $90 - 95\%$ of control loops across diverse domains, ranging from power electronics, pneumatics, to self-driving cars, are implemented as PID [84]. The control law for a PID controller can be defined as

$$u(t) = K_P\ e(t) + K_D\ \dot{e}(t) + K_I \int e(t)\ dt \tag{3.12}$$

where K_P is the proportional term, K_D is the derivative term, and K_I is the integral term. $e(t)$ is the tracking error defined as $e(t) = X_{des}(t) - X_{meas}(t)$, where X_{des} is a desired trajectory that contains a subset of controllable states of the system, and X_{meas} is the measured states. For UAV, the control input $u \in \mathbb{R}^4$, can be the virtual controls, Generalized forces, or the actuator level control rotor velocities. For trajectory tracking in UAVs, the control loop is typically organized into an outer-loop and an inner-loop [85] (Fig. 3.8)

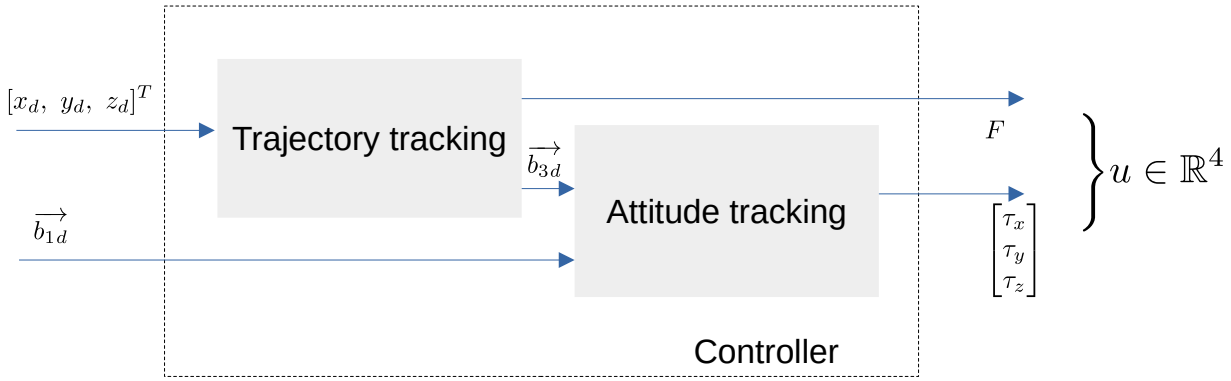

Figure 3.8: The control system structure for UAVs

The outer-loop is concerned with the position tracking and provides a reference value for the desired body-frame z axis as $\overrightarrow{b_{3d}}$, and the inner-loop is concerned with attitude stabilization that leverages the output from the outer-loop as well as a desired body-frame x axis ($\overrightarrow{b_{1d}}$) to compute torques. Typically, we need to ensure that $f_{attitude} > f_{trajectory}$, that is, the frequency of the inner-loop should be higher than that of the outer-loop. This is due to the coupled nature of translational and rotational dynamics of UAVs. Since control force can only be generated along the body-frame z axis, to achieve a desired force direction, the z axis needs to be oriented towards the desired force direction, necessitating the attitude loop to run faster relatively.

Specifically, based on the control system architecture in Fig. 3.8, and the control law in Eq. (3.12), we implement the PID tracking controller for QDrone2 and test it using our control system toolbox. The trajectory is defined as

$$r(t) = \begin{bmatrix} x(t) \\ y(t) \\ z(t) \\ \psi(t) \end{bmatrix} \tag{3.13}$$

where $[x(t),\ y(t),\ z(t)]$ is the desired inertial frame position, and $\psi(t)$ is the desired yaw angle. Note that roll and pitch angles cannot be controlled directly due to the underactuated nature of the system. The reference can be arbitrarily defined; however, here we choose an uneven pulse trajectory along x axis represented as follows

$$\begin{aligned} x(t) &= \begin{cases} 0 & \text{for } 0 \leq t < 30 \\ -0.75 & \text{for } 30 \leq t < 35 \\ 0.75 & \text{for } 35 \leq t < 45 \\ -0.75 & \text{for } 45 \leq t < 50 \\ 0 & \text{for } 50 \leq t \leq 60 \end{cases} \\ z(t) &= \begin{cases} 0 & \text{for } 0 \leq t < 22 \\ 0.75 & \text{for } 22 \leq t \leq 60 \end{cases} \\ r(t) &= \begin{bmatrix} x(t) \\ 0 \\ z(t) \\ 0 \end{bmatrix} \end{aligned} \tag{3.14}$$

The outer-loop design is as follows: given the state vector X, its derivative $\dot{X}$, and a

partially used reference state X_{ref} (see Section 2.2.4 for definition of state)

$$
\begin{aligned}
e_p(t) &= \begin{bmatrix} x(t) \\ 0 \\ z(t) \\ 0 \end{bmatrix} - \begin{bmatrix} {}^I x \\ {}^I y \\ {}^I z \\ \psi \end{bmatrix} \\
e_v(t) &= - \begin{bmatrix} {}^I v_x \\ {}^I v_y \\ {}^I v_z \\ \dot{\psi} \end{bmatrix} \\
u_{outer}(t) &= K_{outer,\ P}\ e_p(t) + K_{outer,\ D}\ e_v(t) + K_{outer,\ I}\ \sum_i e_p(t_i)\Delta t
\end{aligned}
\tag{3.15}
$$

where $\begin{bmatrix} {}^I x & {}^I y & {}^I z \end{bmatrix}^T = X^{(1:3)}$, $\begin{bmatrix} {}^I v_x & {}^I v_y & {}^I v_z \end{bmatrix}^T = X^{(4:6)}$, $\begin{bmatrix} \text{sgn}(\sin \frac{2\pi \cdot t}{T}) & 0 & 0.75 \end{bmatrix}^T = X_{ref}^{(1:3)}$, and Δt is the sampling time of the controller (this will be equivalent to $\frac{1}{f_{control}}$). Note that we used a discrete time integration approximation for the integral error. Moreover, to obtain the yaw angle and its derivative, we use the following relation for converting quaternions to an Euler angle sequence

$$
\begin{bmatrix} \phi \\ \theta \\ \psi \end{bmatrix} = \begin{bmatrix} \arctan \frac{2(q_z q_y + q_w q_x)}{1 - 2(q_x^2 + q_y^2)} \\ \arcsin 2(q_w q_y - q_z q_x) \\ \arctan \frac{2(q_y q_x + q_w q_z)}{1 - 2(q_y^2 + q_z^2)} \end{bmatrix}
\tag{3.16}
$$

Using this relation, we obtain

$$
\psi_{ref} = \frac{2(q_{ref,\ y} q_{ref,\ x} + q_{ref,\ w} q_{ref,\ z})}{1 - 2(q_{ref,\ y}^2 + q_{ref,\ z}^2)}
\tag{3.17}
$$

where $\begin{bmatrix} q_{ref,\ w} & q_{ref,\ x} & q_{ref,\ y} & q_{ref,\ z} \end{bmatrix}^T = X_{ref}^{(7:10)}$ Finally, $\dot{\psi}$ is obtained using Eq. (2.8), after obtaining the Euler angles using Eq. (3.16) with $Q = \begin{bmatrix} q_w & q_x & q_y & q_z \end{bmatrix}^T = X^{(7:10)}$

The outer loop control u_{outer} is transformed into an intermediate form before being passed to the inner loop. This transformation is a simple linear map that permutes the x and y commands. The motivation behind this is that a tracking error along the x-axis requires the drone's body-frame z-axis, ${}^{\mathbf{I}}\mathbf{e}_{\mathbf{3}}^{\mathbf{b}}$, to orient itself toward the inertial x-axis.

Similarly, a tracking error along the y-axis requires alignment of ${}^{\mathbf{I}}\mathbf{e}_{\mathbf{3}}^{\mathbf{b}}$ with the inertial y-axis, due to the constraint that the UAV can only apply thrust along ${}^{\mathbf{I}}\mathbf{e}_{\mathbf{3}}^{\mathbf{b}}$.

However, a positive roll-axis command causes the body-frame vector ${}^{\mathbf{I}}\mathbf{e}_{\mathbf{3}}^{\mathbf{b}}$ to increase its angle with respect to the inertial y-axis, whereas a positive pitch-axis command causes ${}^{\mathbf{I}}\mathbf{e}_{\mathbf{3}}^{\mathbf{b}}$ to decrease its angle with respect to the inertial x-axis. These geometric relationships justify the structure of the mapping matrix.

Finally, we include a gravity compensation term in the thrust command. Using this reasoning, the intermediate mapping matrix is constructed as follows:

$$
\begin{aligned}
{}' u_{outer} &= \begin{bmatrix} T \\ {}'\phi_{des} \\ {}'\theta_{des} \\ {}'\psi_{des} \end{bmatrix} = \begin{bmatrix} T \\ r_{inner}(t) \end{bmatrix} \\
&= \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & -1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} u_{outer} + \begin{bmatrix} mg \\ 0 \\ 0 \\ 0 \end{bmatrix} \\
&= \underbrace{\mathcal{M}}_{mapping\ matrix} u_{outer} + \underbrace{\begin{bmatrix} mg \\ 0 \\ 0 \\ 0 \end{bmatrix}}_{gravity\ compensation}
\end{aligned}
\tag{3.18}
$$

Here, $m = 1.504$ kg is the mass of QDrone2 and $g = 9.81\ \frac{\mathrm{m}}{\mathrm{s}^2}$ is the acceleration due to gravity

The inner-loop control law is defined as follows, with the following tracking errors

$$
\begin{aligned}
e_{attitude}(t) &= r_{inner}(t) - \begin{bmatrix} \phi \\ \theta \\ 0 \end{bmatrix} \\
e_{angular}(t) &= - \begin{bmatrix} {}^{B}\omega_x \\ {}^{B}\omega_y \\ {}^{B}\omega_z \end{bmatrix} \\
u_{inner} &= K_{inner,\ P}\ e_{attitude}(t) + K_{inner,\ D,\ 1}\ e_{angular}(t)) - K_{inner,\ D,\ 2}\ \begin{bmatrix} {}^{B}\alpha_x \\ {}^{B}\alpha_y \\ {}^{B}\alpha_z \end{bmatrix}
\end{aligned}
\tag{3.19}
$$

where, $\begin{bmatrix} {}^{B}\omega_x & {}^{B}\omega_y & {}^{B}\omega_z \end{bmatrix}^T = X^{(11:13)}$, the body frame angular velocity and $\begin{bmatrix} {}^{B}\alpha_x & {}^{B}\alpha_y & {}^{B}\alpha_z \end{bmatrix}^T = \dot{X}^{(11:13)}$ is the angular acceleration of body frame.

The final virtual control inputs, the generalized forces from this controller, are defined as

$$
\begin{aligned}
u(t) &= \begin{bmatrix} T \\ \tau_x \\ \tau_y \\ \tau_z \end{bmatrix} \\
&= \begin{bmatrix} T \\ u_{inner} \end{bmatrix} \in \mathbb{R}^4
\end{aligned}
\tag{3.20}
$$

After defining this control law inside the body of `calculate_u` of our custom controller inherited from `ControllerBase`, we deploy it with the `CustomServer` configured with $f_{physics} = 1000$ Hz, $f_{control} = 1000$ Hz, $RTF = 1$ (for HIL simulation). Since `CustomServer` is stepped by the end user, we execute a thread that performs a single step on the actual Gz server and sleeps for the remaining time required to achieve a $RTF = 1$.

3.6.1.2 PID Controller Simulation Results

The following results are obtained from the simulation. The control gains used are listed below

$$
\begin{aligned}
K_{outer,\ P} &= \mathrm{diag}(\begin{bmatrix} 0.471\ \frac{\mathrm{rad}}{\mathrm{m}} & 0.471\ \frac{\mathrm{rad}}{\mathrm{m}} & 31.500\ \frac{\mathrm{N}}{\mathrm{m}} & 0.075\ \frac{1}{\mathrm{s}} \end{bmatrix}) \\
K_{outer,\ D} &= \mathrm{diag}(\begin{bmatrix} 0.418\ \frac{\mathrm{rad{\cdot}s}}{\mathrm{m}} & 0.418\ \frac{\mathrm{rad{\cdot}s}}{\mathrm{m}} & 18.670\ \frac{\mathrm{N{\cdot}s}}{\mathrm{m}} & 0.002 \end{bmatrix}) \\
K_{outer,\ I} &= \mathrm{diag}(\begin{bmatrix} 0.010\ \frac{\mathrm{rad}}{\mathrm{m{\cdot}s}} & 0.010\ \frac{\mathrm{rad}}{\mathrm{m{\cdot}s}} & 6.000\ \frac{\mathrm{N}}{\mathrm{m{\cdot}s}} & 0\ \frac{1}{\mathrm{s}^2} \end{bmatrix}) \\
K_{inner,\ P} &= \mathrm{diag}(\begin{bmatrix} 12\ \frac{\mathrm{N{\cdot}m}}{\mathrm{rad}} & 12\ \frac{\mathrm{N{\cdot}m}}{\mathrm{rad}} & 0.020\ \frac{\mathrm{N{\cdot}m{\cdot}s}}{\mathrm{rad}} \end{bmatrix}) \\
K_{inner,\ D,\ 1} &= \mathrm{diag}(\begin{bmatrix} 0.100\ \frac{\mathrm{N{\cdot}m{\cdot}s}}{\mathrm{rad}} & 0.100\ \frac{\mathrm{N{\cdot}m{\cdot}s}}{\mathrm{rad}} & 0.0\ \frac{\mathrm{N{\cdot}m{\cdot}s}^2}{\mathrm{rad}} \end{bmatrix}) \\
K_{inner,\ D,\ 2} &= \mathrm{diag}(\begin{bmatrix} 0.003\ \frac{\mathrm{N{\cdot}m{\cdot}s}^2}{\mathrm{rad}} & 0.003\ \frac{\mathrm{N{\cdot}m{\cdot}s}^2}{\mathrm{rad}} & 0.001\ \frac{\mathrm{N{\cdot}m{\cdot}s}^3}{\mathrm{rad}} \end{bmatrix})
\end{aligned}
\tag{3.21}
$$

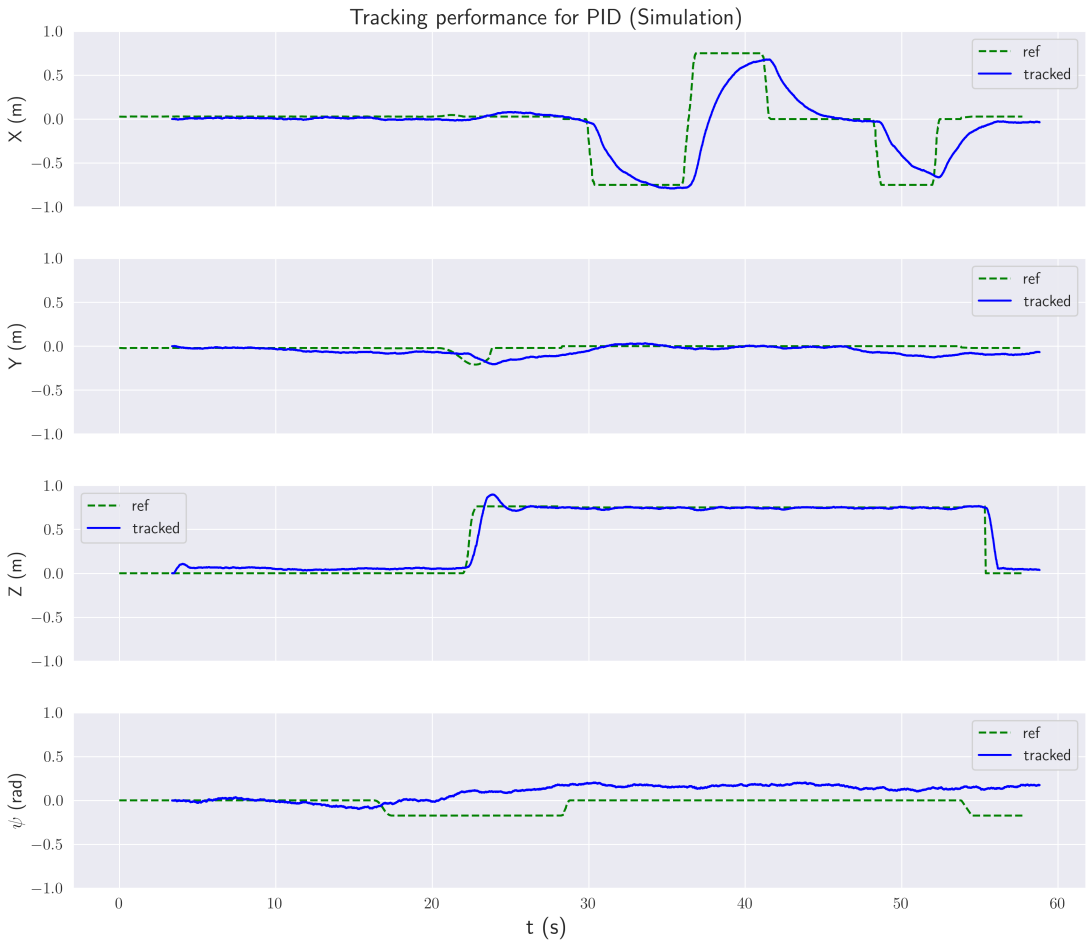

Figure 3.9: Tracking performance in x-y-z-ψ with PID controller in simulation

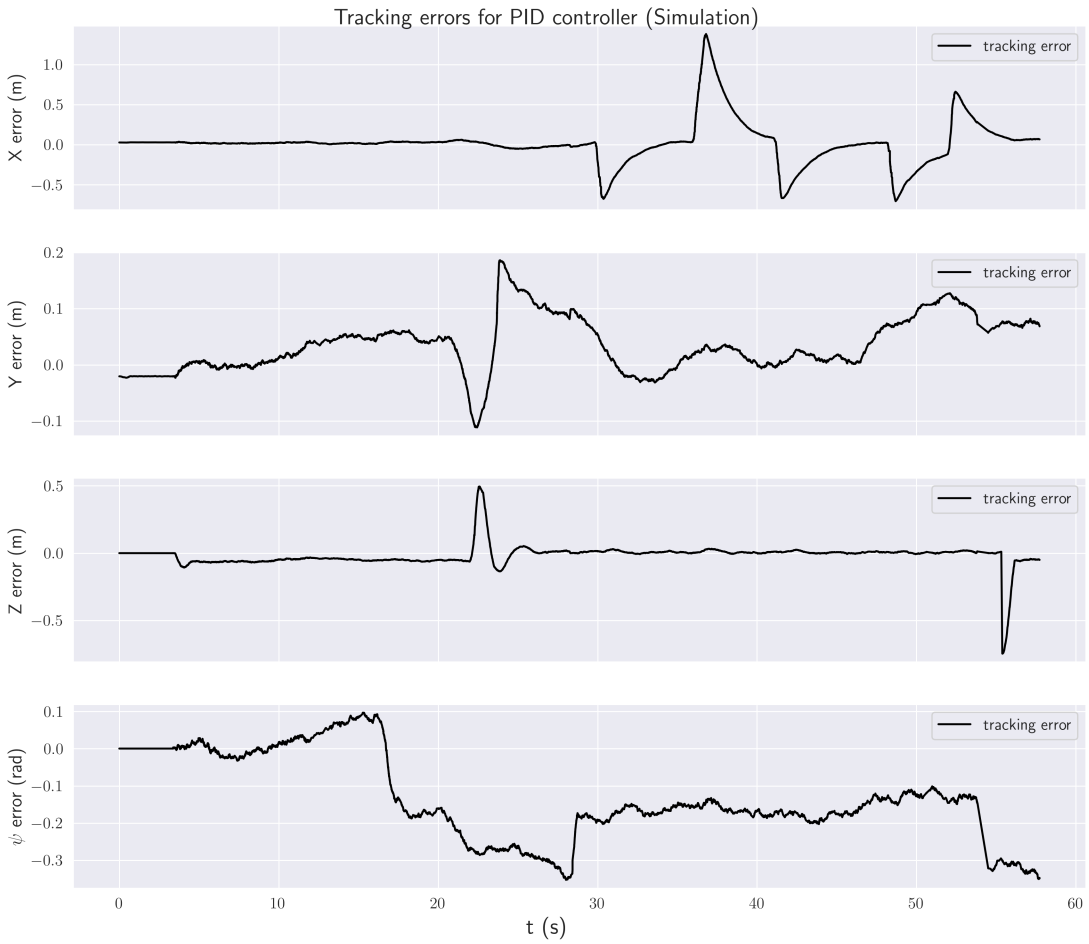

Figure 3.10: Tracking errors with PID controller in simulation.

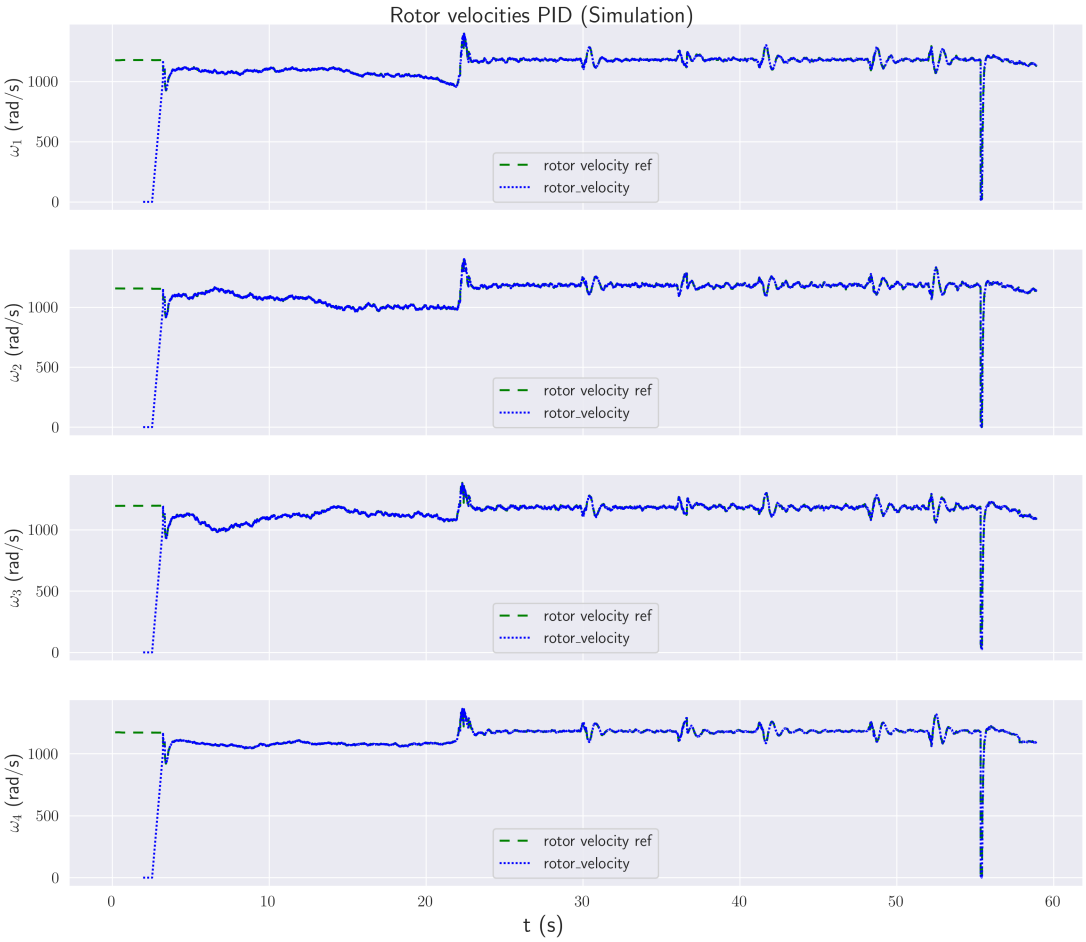

Figure 3.11: Commanded rotor velocities and the achieved rotor velocities with PID controller in simulation.

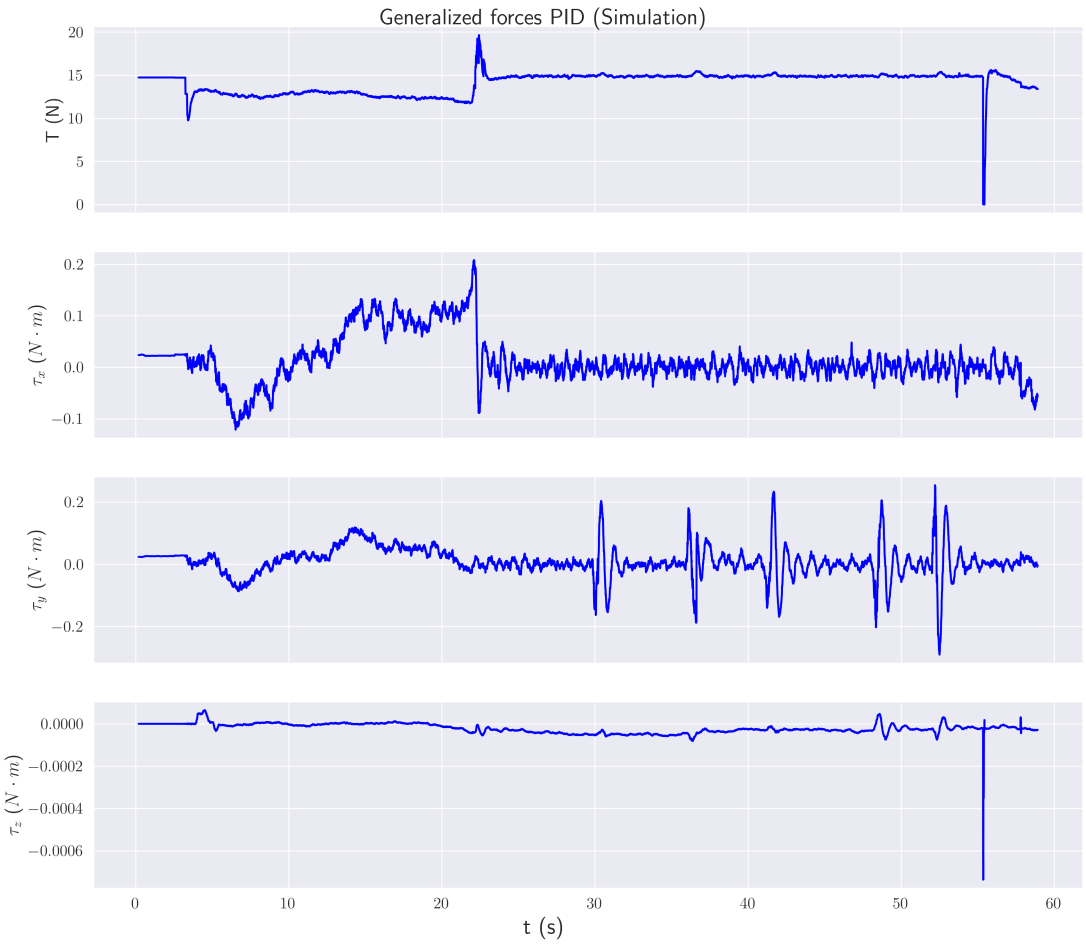

Figure 3.12: Commanded virtual controls, Generalized forces, with PID controller in simulation.

Table 3.1: Tracking RMSE for x-y-z-ψ

State	**RMSE**
x	0.1213 m
y	0.0385 m
z	0.0377 m
ψ	0.1179 rad

3.6.1.3 PID Controller Hardware Results

After validating the controller in simulation, we conducted hardware experiments with the same reference trajectory on QDrone2. The PID controller was implemented using the QUARC software in Simulink. The controller runs at 1000 Hz, similar to the simulation. The feedback is obtained using the OptiTrack Motion Capture system, which is configured with 12 cameras. The gains we use are

$$\begin{aligned}
K_{outer,\ P} &= \text{diag}(\begin{bmatrix} 0.524 \cdot \frac{\text{rad}}{\text{m}} & 0.524 \cdot \frac{\text{rad}}{\text{m}} & 35.000 \cdot \frac{\text{N}}{\text{m}} & 1.5 \cdot \frac{1}{\text{s}} \end{bmatrix}) \\
K_{outer,\ D} &= \text{diag}(\begin{bmatrix} 0.418 \cdot \frac{\text{rad}\cdot\text{s}}{\text{m}} & 0.418 \cdot \frac{\text{rad}\cdot\text{s}}{\text{m}} & 18.000 \cdot \frac{\text{N}\cdot\text{s}}{\text{m}} & 0.4 \end{bmatrix}) \\
K_{outer,\ I} &= \text{diag}(\begin{bmatrix} 0.010 \cdot \frac{\text{rad}}{\text{m}\cdot\text{s}} & 0.010 \cdot \frac{\text{rad}}{\text{m}\cdot\text{s}} & 6.000 \cdot \frac{\text{N}}{\text{m}\cdot\text{s}} & 0. \cdot \frac{1}{\text{s}^2} \end{bmatrix}) \\
K_{inner,\ P} &= \text{diag}(\begin{bmatrix} 12 \cdot \frac{\text{N}\cdot\text{m}}{\text{rad}} & 12 \cdot \frac{\text{N}\cdot\text{m}}{\text{rad}} & 2.0 \cdot \frac{\text{N}\cdot\text{m}\cdot\text{s}}{\text{rad}} \end{bmatrix}) \\
K_{inner,\ D,\ 1} &= \text{diag}(\begin{bmatrix} 0.100 \cdot \frac{\text{N}\cdot\text{m}\cdot\text{s}}{\text{rad}} & 0.100 \cdot \frac{\text{N}\cdot\text{m}\cdot\text{s}}{\text{rad}} & 0.01 \cdot \frac{\text{N}\cdot\text{m}\cdot\text{s}^2}{\text{rad}} \end{bmatrix}) \\
K_{inner,\ D,\ 2} &= \text{diag}(\begin{bmatrix} 0.003 \cdot \frac{\text{N}\cdot\text{m}\cdot\text{s}^2}{\text{rad}} & 0.003 \cdot \frac{\text{N}\cdot\text{m}\cdot\text{s}^2}{\text{rad}} & 0.001 \cdot \frac{\text{N}\cdot\text{m}\cdot\text{s}^3}{\text{rad}} \end{bmatrix})
\end{aligned} \tag{3.22}$$

Note that the gains are very similar to the simulation values.

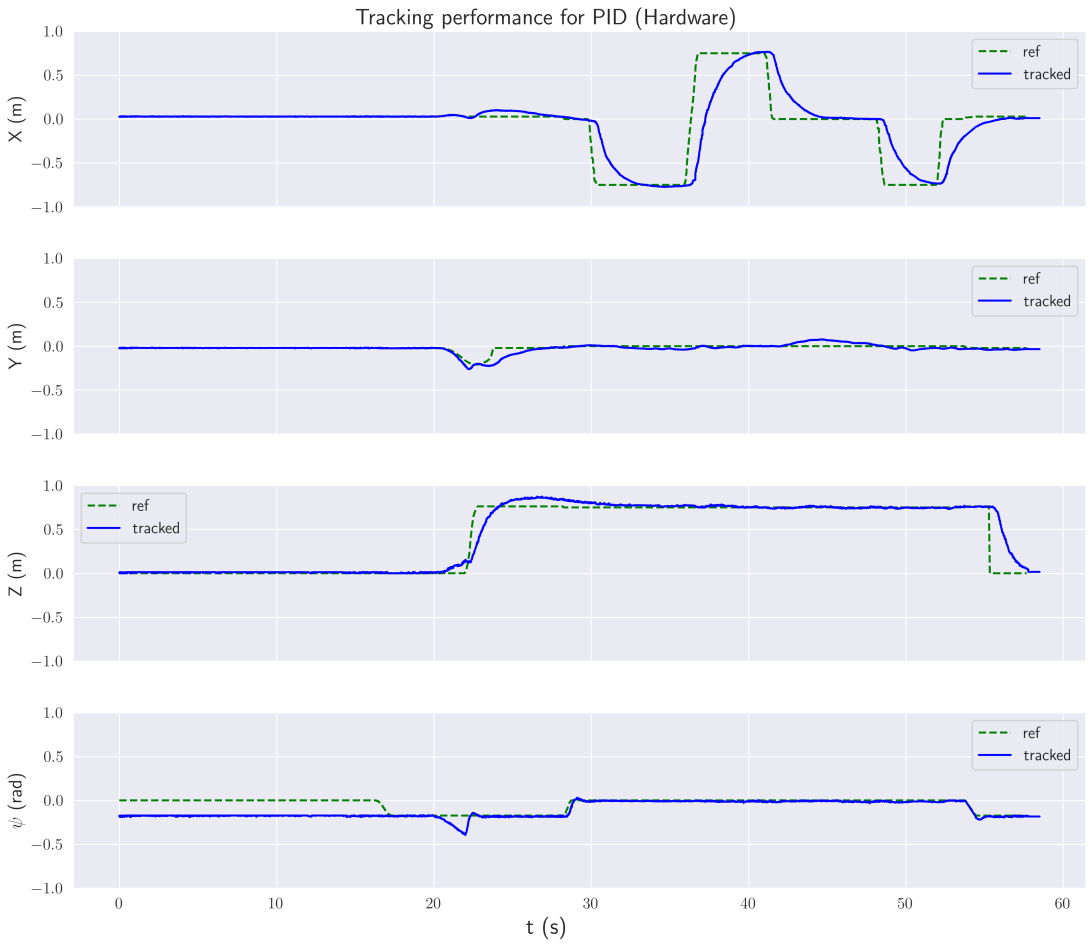

Figure 3.13: Tracking performance in x-y-z-ψ with PID controller with hardware

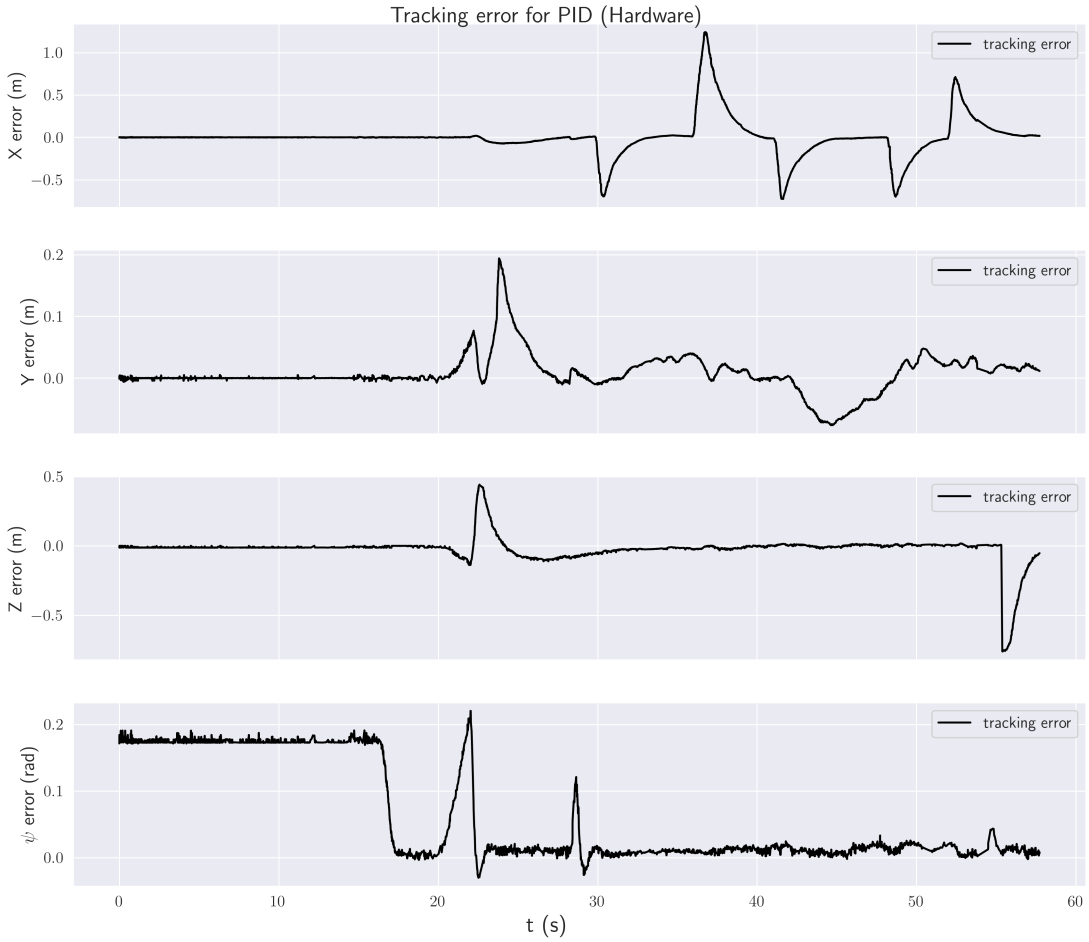

Figure 3.14: Tracking errors in x-y-z-ψ with PID controller with hardware

Table 3.2: Tracking RMSE for x-y-z-ψ with hardware

State	**RMSE**
x	0.085 m
y	0.014 m
z	0.036 m
ψ	0.087 rad

Note that the RMSE in simulation (Table. 3.3) is higher compared to hardware, this is justified by the additive noise injection.

Table 3.3: Tracking RMSE comparison for x-y-z-ψ

height**State**	**RMSE-Simulation**	**RMSE-Hardware**
x, m	0.1213	0.085
y, m	0.0385	0.014
z, m	0.0377	0.036
ψ, rad	0.1179	0.087

3.6.1.4 NMPC using Open-source OCP Solver

In this example, we demonstrate the integration of open-source packages to test control systems for UAVs, another feature of our toolbox.

Acados [86]-CasADi [87] is a popular open-source suite for Optimal Control Problem solvers in embedded applications. Leveraging their software, we implement an NMPC for the outer-loop control of a quadrotor and test the controller using our toolbox. We use the previous definition of the state vector X. The simplified rigid body dynamics of state $\dot{X}$ can be written as follows, with $\omega := {}^B\omega = \begin{bmatrix} {}^B\omega_x & {}^B\omega_y & {}^B\omega_z \end{bmatrix}^T$ for the remainder of this section,

$$\dot{X} = f(X, u) = \begin{bmatrix} {}^I v_x \\ {}^I v_y \\ {}^I v_z \\ \frac{\overrightarrow{T}}{m} \\ \frac{1}{2}\Psi(Q)\bar{\omega} \\ J^{-1}(\overrightarrow{M} - \omega \times J\omega) \end{bmatrix} \tag{3.23}$$

Here $\overrightarrow{T} = {}^B_I R \begin{bmatrix} 0 \\ 0 \\ ({}^B_I R \mathbf{e_3}) \cdot \begin{bmatrix} u_1 & u_2 & u_3 \end{bmatrix}^T \end{bmatrix} - mg\mathbf{e_3}$, $\Psi(Q)$ and $\bar{\omega}$ are defined in Section 2.2.2.3, and $\overrightarrow{M} = \begin{bmatrix} u_4 & u_5 & u_6 \end{bmatrix}^T$. The control input optimized by the NMPC is $u = \begin{bmatrix} u_1 & u_2 & u_3 & u_4 & u_5 & u_6 \end{bmatrix}^T$, however, we only use $\begin{bmatrix} u_1 & u_2 & u_3 \end{bmatrix}^T$ from MPC for tracking, making it an outer-loop controller, and leveraging the previously implemented Geometric controller for attitude tracking. This is motivated by the fact that OCP solving

can be computationally expensive, and as stated earlier, we require the inner loop to be executed faster. Referring to Fig. 3.8, we can see that the outer loop commands are independent of the inner loop, therefore we can safely replace the inner loop controller with another controller.

To implement NMPC using Acados-CasADi, we first define the simplified dynamics using the CasADi symbolic library, followed by an OCP solver object creation using Acados. The solver will be used as our outer-loop controller, and we will use the Geometric controller to track the inner loop. The cost matrices Q for the state, and R for the control are defined as

$$\begin{aligned} Q &= \text{diag}(\begin{bmatrix} \mathbf{1}^{6\times1} & \mathbf{1}^{4\times1} \times 0.1 & \mathbf{1}^{3\times1} \end{bmatrix}) \\ R &= \text{diag}(\begin{bmatrix} \mathbf{1}^{3\times1} \times 0.3 & \mathbf{1}^{3\times1} \times 0.03 \end{bmatrix}) \end{aligned} \tag{3.24}$$

The prediction horizon we use is $N = 100$, with a sampling time of $T = 0.01$ s (100 Hz), allowing the model prediction across 1 s. We track an infinity-shaped trajectory parametrized as follows

$$\begin{aligned} p_{des} &= \begin{bmatrix} 0.8\sin(\frac{2\pi t}{60})) \\ 0.6\sin(\frac{4\pi t}{60})) \\ 1.2 \end{bmatrix} \\ v_{des} = \dot{p}_{des} &= \begin{bmatrix} \frac{1.6\pi}{60}\cos(\frac{2\pi t}{60})) \\ \frac{2.4\pi}{60}\cos(\frac{4\pi t}{60})) \\ 0 \end{bmatrix} \end{aligned} \tag{3.25}$$

The tracking results in simulation is shown below

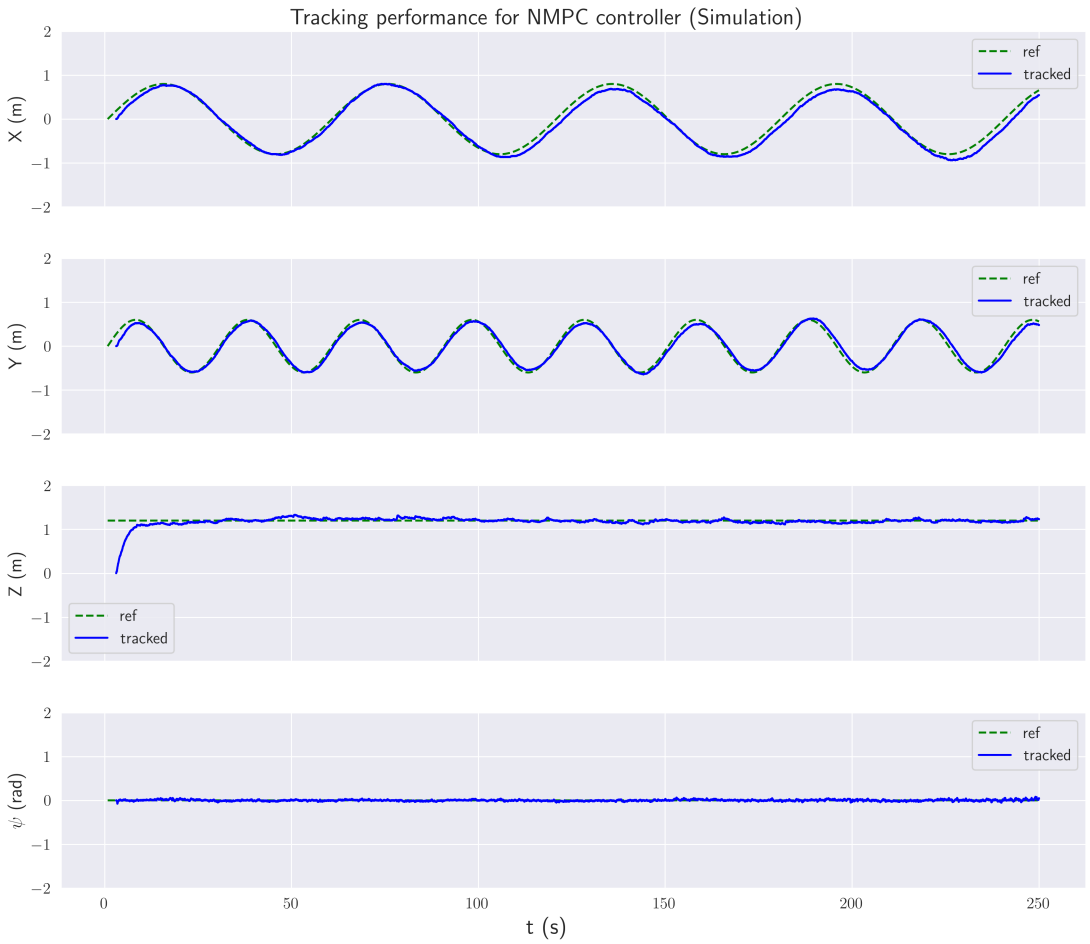

Figure 3.15: Tracking performance in x-y-z-ψ with NMPC in simulation

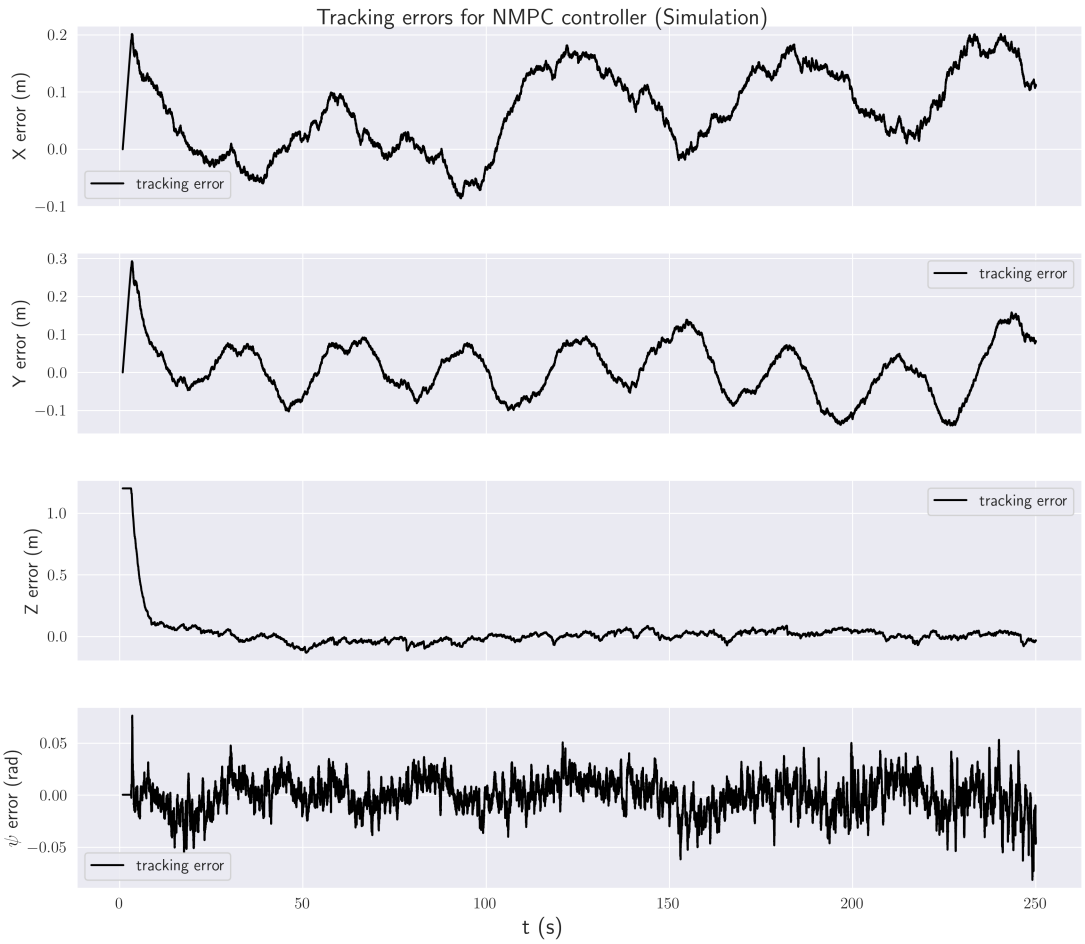

Figure 3.16: Tracking errors in x-y-z-ψ with NMPC controller in simulation

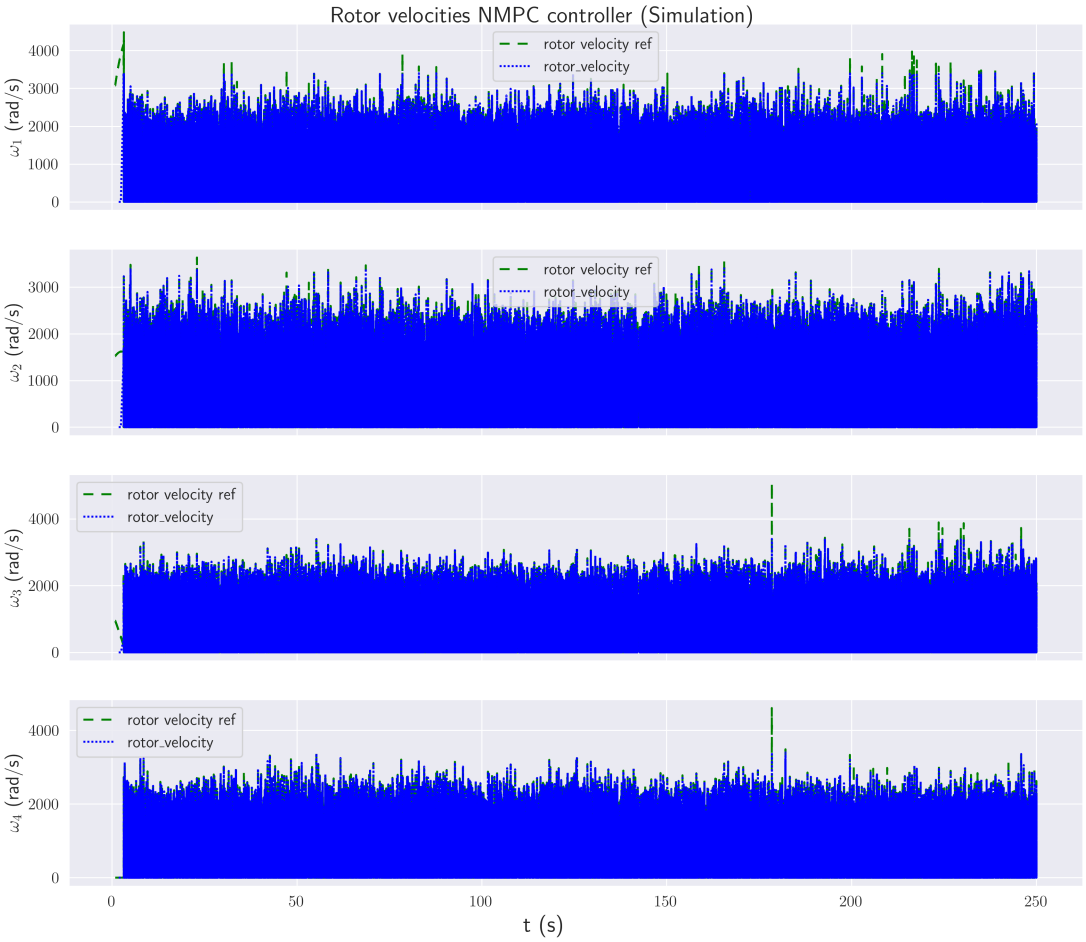

Figure 3.17: Commanded rotor velocities and the achieved rotor velocities with the NMPC controller in simulation

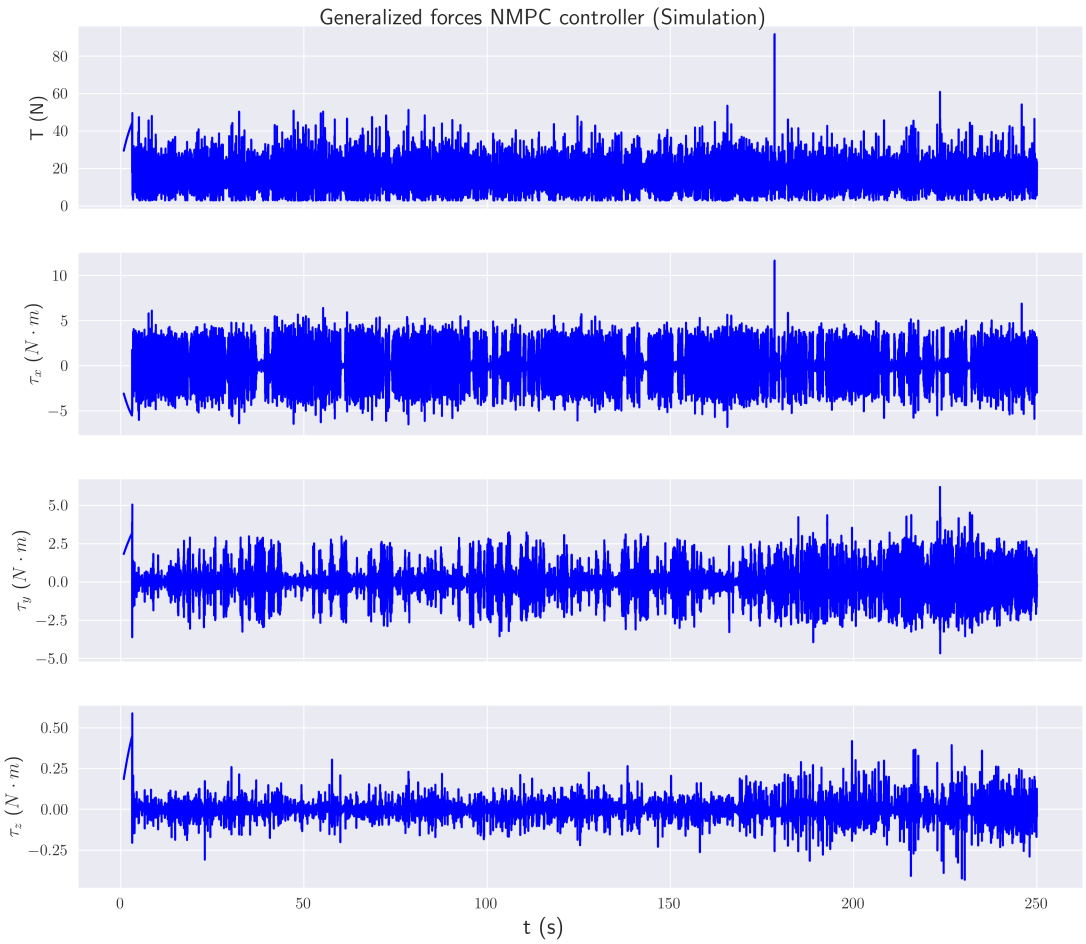

Figure 3.18: Virtual controls, the generalized forces, computed with the NMPC controller in simulation

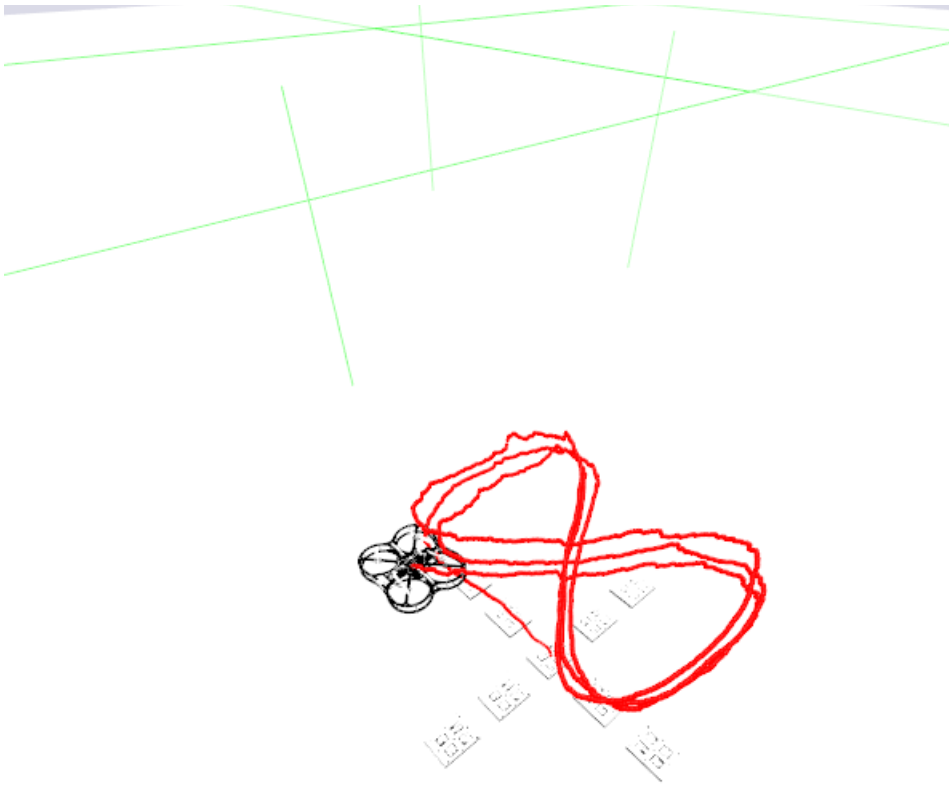

Figure 3.19: UAV tracking infinity trajectory in Gazebo

Table 3.4: Tracking RMSE for x-y-z-ψ with NMPC controller

State	**RMSE**
x	0.080 m
y	0.060 m
z	0.0355 m
ψ	0.012 rad

3.6.1.5 NMPC Hardware results

We deploy the Acados-CasADi NMPC controller in the QDrone2 hardware using the Simulink bridge provided by Acados software. The cost matrices Q and R are kept the same, and the results are shown below

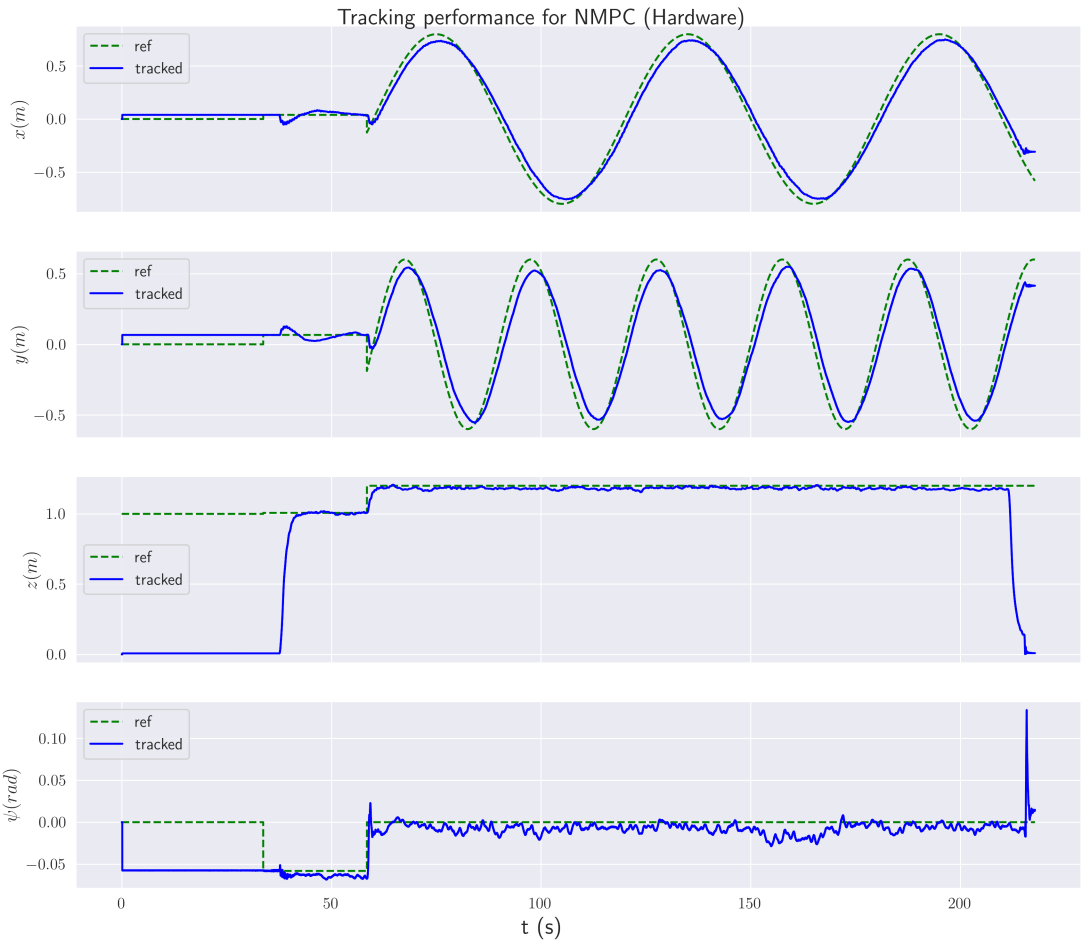

Figure 3.20: Tracking performance in x-y-z-ψ with NMPC controller on hardware

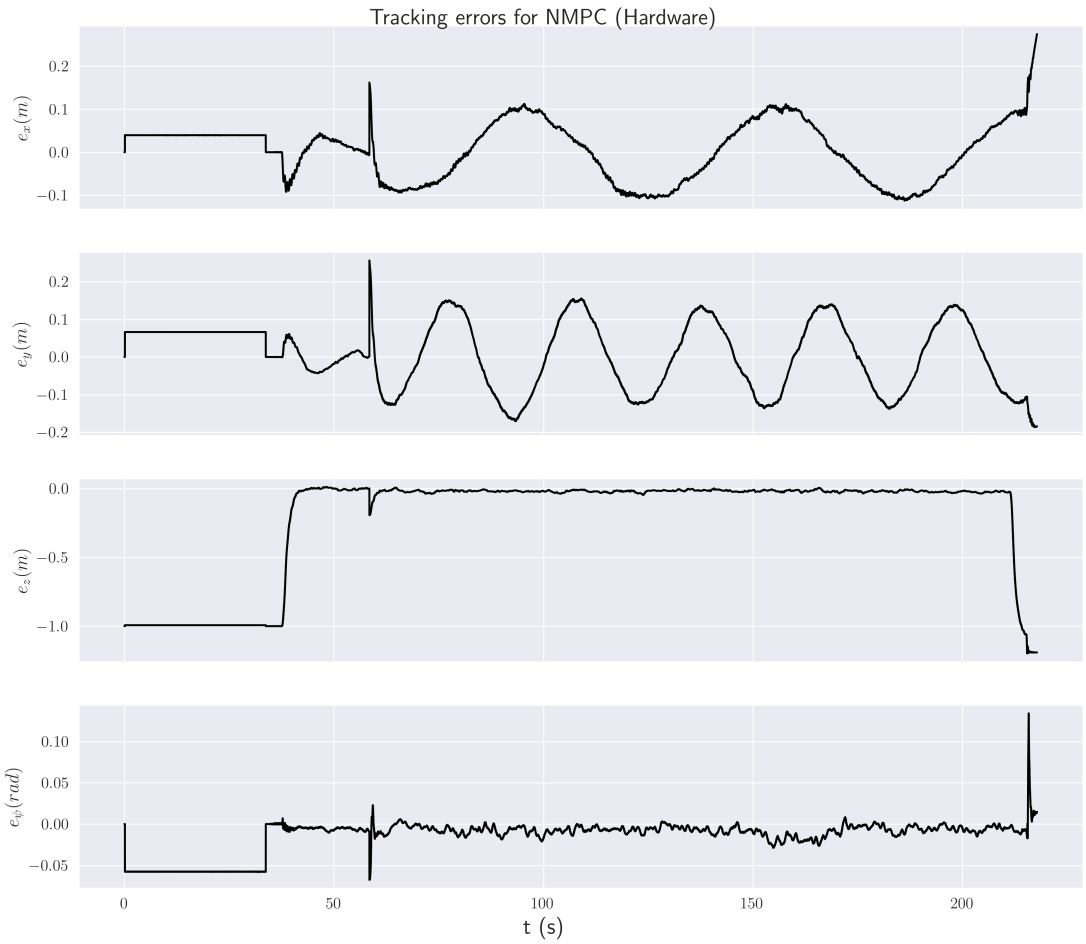

Figure 3.21: Tracking errors in x-y-z-ψ with NMPC controller on hardware

Table 3.5: Tracking RMSE for x-y-z-ψ with NMPC on hardware

State	**RMSE**
x	0.057 m
y	0.077 m
z	0.219 m
ψ	0.015 rad

Table 3.6: Tracking RMSE comparison for x-y-z-ψ

State	**RMSE-Simulation**	**RMSE-Hardware**
x, m	0.080	0.057
y, m	0.060	0.077
z, m	0.0355	0.219
ψ, rad	0.012	0.015

3.6.2 Comparison of PID and NMPC

The comparison between PID and NMPC controllers in both simulation and hardware are shown here.

State	**RMSE - Simulation**		**RMSE - Hardware**	
	PID	NMPC	PID	NMPC
x, m	0.1213	0.080	0.085	0.057
y, m	0.0385	0.060	0.014	0.077
z, m	0.0377	0.0355	0.036	0.219
ψ, rad	0.1179	0.012	0.087	0.015

Table 3.7: RMSE comparison of PID and NMPC in simulation and hardware

3.6.3 Extended Applications

Here, we showcase simulation-only applications using our control library.

3.6.3.1 Geometric Tracking Controller in $SE(3)$

The majority of the controllers used for UAVs are PID controllers, which are linear. However, they have several limitations, which include employing the Euler angles for feedback, which have a singularity when performing complex rotational maneuvers [85]. Geometric Controllers are non-linear controllers leveraging Geometric mechanics. Specifically, geometric mechanics can be used to define dynamic systems evolving on a non-linear manifold,

such as $SE(3)$, the semi-direct product of $SO(3)$ and $\mathbb{R}^3$, precisely, $SE(3) = SO(3) \times \mathbb{R}^3$. $SE(3)$ is a Lie group, that is, a smooth manifold that satisfies the axioms of a group, enabling application of calculus using the local properties of a manifold, and the application of the non-linear composition property of groups [88].

In this application, we demonstrate the implementation of the non-linear geometric tracking controller for trajectory tracking of UAVs. Specifically, we design the controller assuming a simplified rigid body dynamics defined as follows

$$\begin{aligned} p &= \begin{bmatrix} {}^I x \\ {}^I y \\ {}^I z \end{bmatrix} \\ \dot{p} &= \begin{bmatrix} {}^I v_x \\ {}^I v_y \\ {}^I v_z \end{bmatrix} \\ m\ddot{p} &= -mg\mathbf{e_3} + {}^B_I R f \mathbf{e3} \\ \dot{R} &= R\hat{\omega} \\ \dot{\omega} &= J^{-1}(M - \omega \times J \cdot \omega) \end{aligned} \tag{3.26}$$

Here, f is the thrust generated along $\mathbf{e_3}$ of the UAV, and M is the total moments in the body frame. Note that we do not use the Euler angles for attitude; rather, we leverage $R \in SO(3)$ to represent the orientation. Since the position $p \in \mathbb{R}^3$, the configuration of the UAV can be represented in $SE(3)$. Furthermore, for the remainder of this section, $\omega := {}^B\omega = [{}^B\omega_x,\ {}^B\omega_y,\ {}^B\omega_z]^T$ and $\hat{\omega}$ is the skew symmetric matrix transform of ω. The control law is defined as follows [85]

$$\begin{aligned} f &= (K_{P,\ outer}\ e_p + K_{D,\ outer}\ e_{\dot{p}} + m\ \ddot{p}_{des}) {}^B_I R \mathbf{e_3} \\ M &= -K_{P,\ inner}\ e_R - K_{D,\ inner}\ e_\omega + \omega \times J\omega \\ &\quad - J(\hat{\omega} {}^B_I R^T {}^B_I R_{des} \omega_{des} - {}^B_I R^T {}^B_I R_{des} \dot{\omega}_{des}) \end{aligned} \tag{3.27}$$

Here, $e_R = \frac{1}{2}({}^B_I R^T_{des} {}^B_I R - {}^B_I R^T {}^B_I R_{des})^\vee$, $e_\omega = \omega - {}^B_I R^T {}^B_I R_{des} \omega_{des}$, $e_p = p_{des} - p$, and

$e_{\dot{p}} = \dot{p}_{des} - \dot{p}$. The trajectory tracking errors have similar definitions to those in the previous sections.

We define the reference trajectory as

$$\begin{aligned}
p_{des} &= \begin{bmatrix} 2\cos(\frac{2\cdot\pi t}{7}) \\ 2\sin(\frac{2\cdot\pi t}{7}) \\ 1 \end{bmatrix} \\
\dot{p}_{des} &= \begin{bmatrix} \frac{-4\pi}{7}\sin(\frac{2\pi t}{7}) \\ \frac{4\pi}{7}\cos(\frac{2\pi t}{7}) \\ 0 \end{bmatrix} \\
\ddot{p}_{des} &= \begin{bmatrix} \frac{-8\pi^2}{49}\cos(\frac{2\pi t}{7}) \\ \frac{-8\pi^2}{49}\sin(\frac{2\pi t}{7}) \\ 0 \end{bmatrix} \\
\omega_{des} &= \mathbf{0^{3\times 1}} \\
\dot{\omega}_{des} &= \mathbf{0^{3\times 1}} \\
b_{1,\,des} &= \begin{bmatrix} \cos\psi & \sin\psi & 0 \end{bmatrix}^T |\psi = 0 \\
&= \begin{bmatrix} 1 & 0 & 0 \end{bmatrix}^T \\
b_{3,\,des} &= \frac{(K_{P,\,outer}\ e_p + K_{D,\,outer}\ e_{\dot{p}} + m\ddot{p}_{des})}{||(K_{P,\,outer}\ e_p + K_{D,\,outer}\ e_{\dot{p}} + m\ddot{p}_{des})||} \\
b_{2,\,des} &= \frac{b_{3,\,des} \times b_{1,\,des}}{||b_{3,\,des} \times b_{1,\,des}||} \\
{}^B_I R_{des} &= \begin{bmatrix} b_{2,\,des} \times b_{3,\,des} & b_{2,\,des} & b_{3,\,des} \end{bmatrix}
\end{aligned} \tag{3.28}$$

The control gains are

$$\begin{aligned}
K_{outer,\,P} &= \mathrm{diag}(\begin{bmatrix} 33.5757 & 33.5757 & 33 \end{bmatrix})\ \frac{\mathrm{N}}{\mathrm{m}} \\
K_{outer,\,D} &= \mathrm{diag}(\begin{bmatrix} 5.7746 & 6.0022 & 14.5 \end{bmatrix})\ \frac{\mathrm{N\cdot s}}{\mathrm{m}} \\
K_{inner,\,P} &= \mathrm{diag}(\begin{bmatrix} 20.75 & 20.75 & 1 \end{bmatrix})\ \cdot \mathrm{Nm} \\
K_{inner,\,D} &= \mathrm{diag}(\begin{bmatrix} 4.5 & 4.5 & 0.05 \end{bmatrix})\ \cdot \frac{\mathrm{Nm\cdot s}}{\mathrm{rad}}
\end{aligned} \tag{3.29}$$

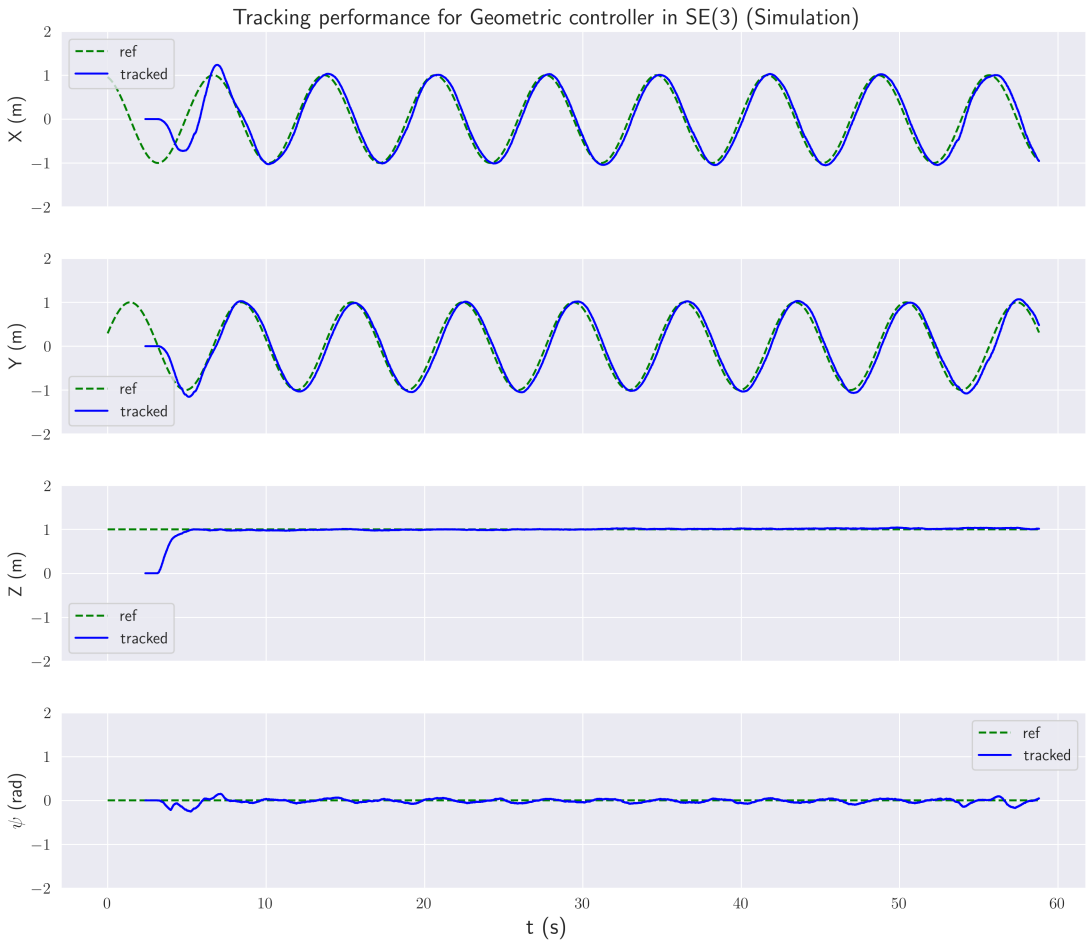

Figure 3.22: Tracking performance in x-y-z-ψ with geometric controller in simulation

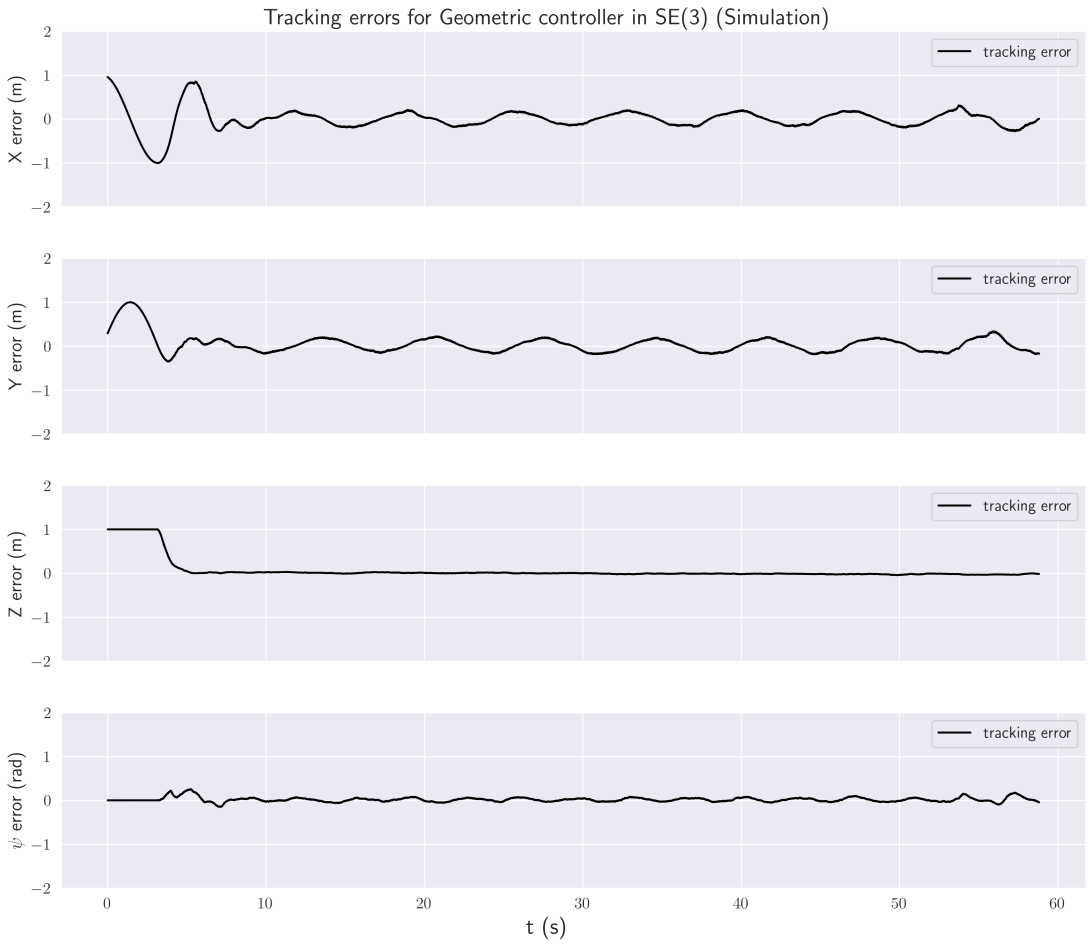

Figure 3.23: Tracking errors in x-y-z-ψ with geometric controller in simulation

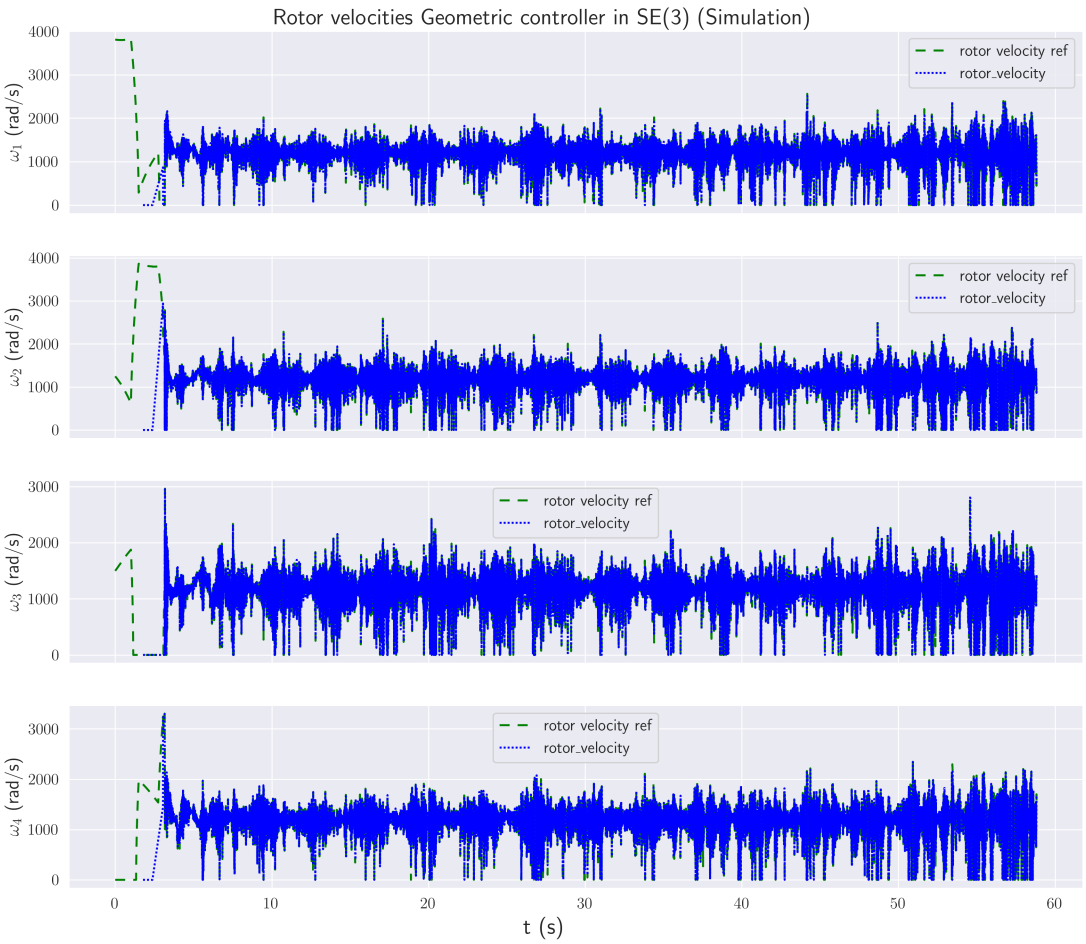

Figure 3.24: Commanded rotor velocities and the achieved rotor velocities with the geometric controller in simulation

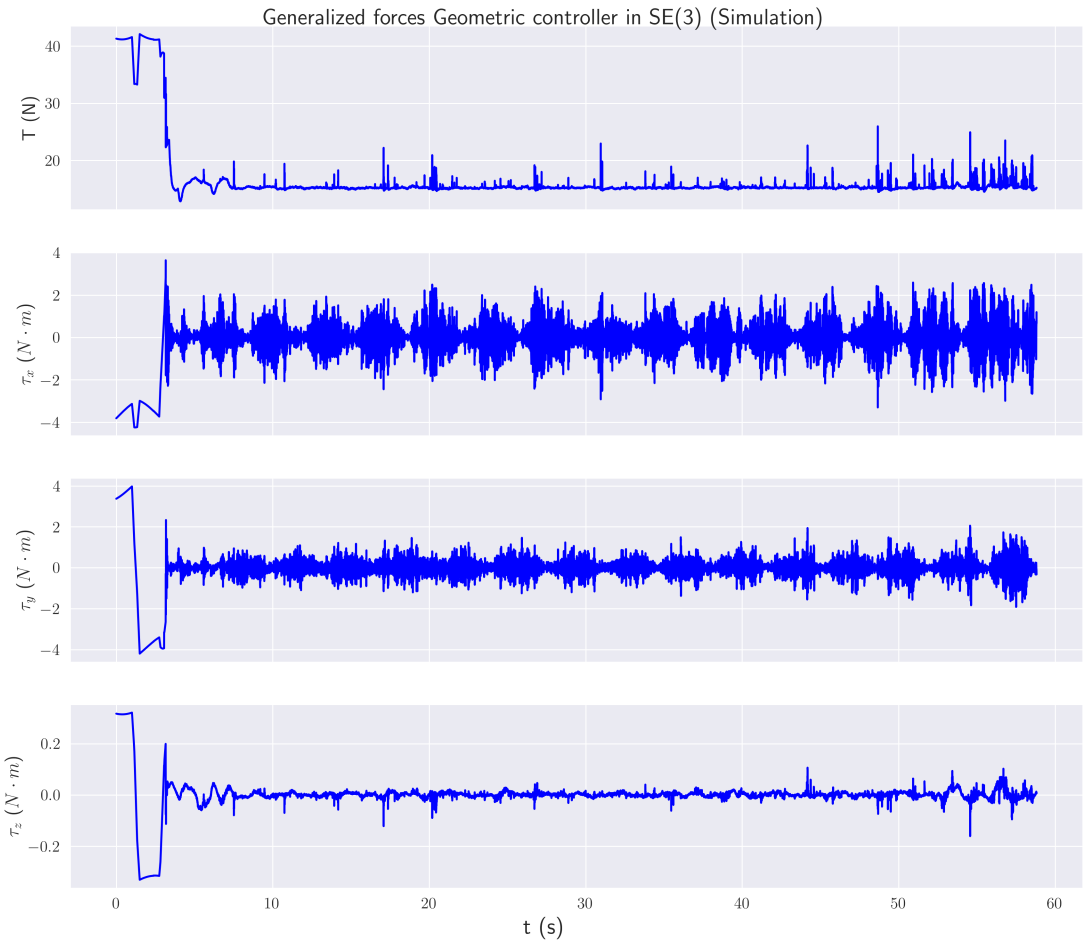

Figure 3.25: Virtual controls, the generalized forces, computed with the geometric controller in simulation

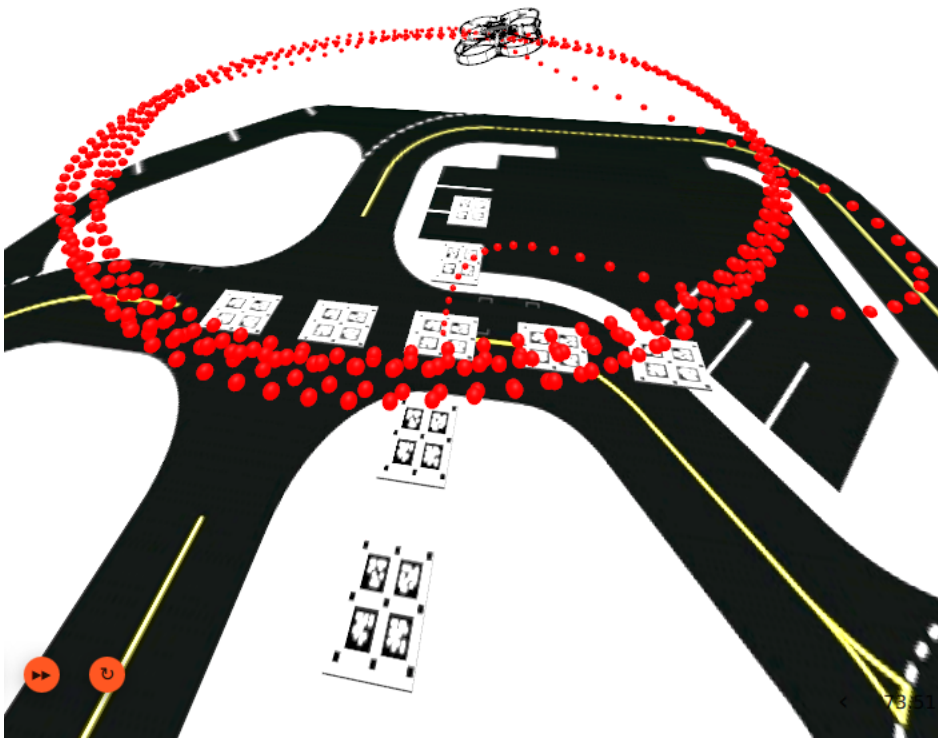

Figure 3.26: Trajectory traced in Gazebo sim

Table 3.8: Tracking RMSE for x-y-z-ψ using geometric controller

State	**RMSE**
x	0.1594 m
y	0.14099 m
z	0.0835 m
ψ	0.0347 rad

3.6.3.2 UAV with Slung Payload

Next, we illustrate the simulation of a more interesting application, a UAV with a slung payload. Slung payload, or tethered payload, is an approach to transporting payload using a UAV where the object to be transported is attached to the UAV system through a tether. This has several applications in the real world, including payload delivery and scientific data collection. However, note that the dynamics of the UAV are affected by the addition of the tether and payload. This necessitates the design of control systems taking into account the changed dynamics.

The state vector for the UAV-Slung-Payload system is

$$X_{slung} = \begin{bmatrix} p \\ s \end{bmatrix} \tag{3.30}$$

where

$$\begin{aligned} p &= \begin{bmatrix} {}^{I}x \\ {}^{I}y \\ {}^{I}z \end{bmatrix} \\ s &= \begin{bmatrix} \gamma_1 \\ \gamma_2 \end{bmatrix} \end{aligned} \tag{3.31}$$

Here, γ_1, γ_2 are the payload swing angles. The dynamics equations after adding a tethered payload are as follows [89]

$$M(X_{slung})\ddot{X}_{slung} + C(X_{slung}, \dot{X}_{slung})\dot{X}_{slung} + G(X_{slung}) = u \tag{3.32}$$

Here, $M(X_{slung})$ is the symmetric mass matrix, $C(X_{slung}, \dot{X}_{slung})$ is the Coriolis matrix, and $G(X_{slung})$ is the gravity compensation. They are defined as follows:

$$M(X_{slung}) = \begin{bmatrix} m_q + m_p & 0 & 0 & m_p l \cos\gamma_1 \cos\gamma_2 & -m_p l \sin\gamma_1 \sin\gamma_2 \\ 0 & m_q + m_p & 0 & 0 & m_p l \cos\gamma_2 \\ 0 & 0 & m_q + m_p & m_p l \sin\gamma_1 \cos\gamma_2 & m_p l \cos\gamma_1 \sin\gamma_2 \\ m_p l \cos\gamma_1 \cos\gamma_2 & 0 & m_p l \sin\gamma_1 \cos\gamma_2 & m\gamma_1 p l^2 \cos{\gamma_2}^2 & 0 \\ -m_p l \sin\gamma_1 \sin\gamma_2 & m_p l \cos\gamma_2 & m_p l \cos\gamma_1 \sin\gamma_2 & 0 & m_p l^2 \end{bmatrix} \tag{3.33}$$

$C(X_{slung},\ \dot{X}_{slung}) =$

$$\begin{bmatrix} 0 & 0 & 0 & -m_p l(\dot{\gamma_1}\sin\gamma_1\cos\gamma_2 + \dot{\gamma_2}\cos\gamma_1\sin\gamma_2) & m_p l(\dot{\gamma_1}\cos\gamma_1\sin\gamma_2 + \dot{\gamma_2}\sin\gamma_1\cos\gamma_2) \\ 0 & 0 & 0 & 0 & -m_p l\dot{\gamma_2}\sin\gamma_2 \\ 0 & 0 & 0 & m_p l(\dot{\gamma_1}\cos\gamma_1\cos\gamma_2 - \dot{\gamma_2}\sin\gamma_1\sin\gamma_2) & m_p l(\dot{\gamma_2}\cos\gamma_1\cos\gamma_2 - \dot{\gamma_1}\sin\gamma_1\sin\gamma_2) \\ 0 & 0 & 0 & -m_p l^2\dot{\gamma_2}\cos\gamma_2\sin\gamma_2 & m_p l^2\dot{\gamma_1}\cos\gamma_2\sin\gamma_2 \\ 0 & 0 & 0 & m_p l^2\dot{\gamma_1}\cos\gamma_2\sin\gamma_2 & 0 \end{bmatrix} \tag{3.34}$$

$$G(X_{slung}) = \begin{bmatrix} 0 \\ 0 \\ 0 \\ m_p g_l \sin\gamma_1\cos\gamma_2 \\ m_p g l\cos\gamma_1\sin\gamma_2 \end{bmatrix} \tag{3.35}$$

where, m_q is the mass of quadrotor, m_p is mass of payload, l is the tether length, and g is the gravity.

We use our `ServerInterface` to obtain the swinging angle states from the Gazebo simulation and then utilize the previously defined Geometric tracking controller to track a similar trajectory to the one in the last section. To obtain the swinging angles from Gazebo using our plugin, we use the following equations using the states obtained for the payload and quadrotor

$$\begin{aligned} \overrightarrow{l} &= \begin{bmatrix} {}^I x_l \\ {}^I y_l \\ {}^I z_l \end{bmatrix} - p \\ \hat{\mathbf{l}} &= \frac{\overrightarrow{l}}{||\overrightarrow{l}||} = \begin{bmatrix} l_1 \\ l_2 \\ l_3 \end{bmatrix} \\ \gamma_1 &= \arctan\frac{l_1}{l_3} \\ \gamma_2 &= \arctan\frac{l_2}{l_3} \\ \dot{\gamma_1} &= \frac{1}{1+\frac{l_1^2}{l_3^2}}(\frac{\dot{l_1}}{l_3} - \frac{\dot{l_3}l_1}{l_3^2}) \\ \dot{\gamma_2} &= \frac{1}{1+\frac{l_2^2}{l_3^2}}(\frac{\dot{l_2}}{l_3} - \frac{\dot{l_3}l_2}{l_3^2}) \end{aligned} \tag{3.36}$$

The terms $\dot{l_2}$ and $\dot{l_1}$ are obtained through finite-difference method as

$$\dot{f}(t) = \frac{f(t) - f(t-T)}{T} \tag{3.37}$$

with $T = \frac{1}{f_{physics}}$ The UAV with the payload is shown in Fig. 3.32.

We use $m_p = 0.500$ kg for the simulation, with the control gains the same as previous section. The tracking results are shown below

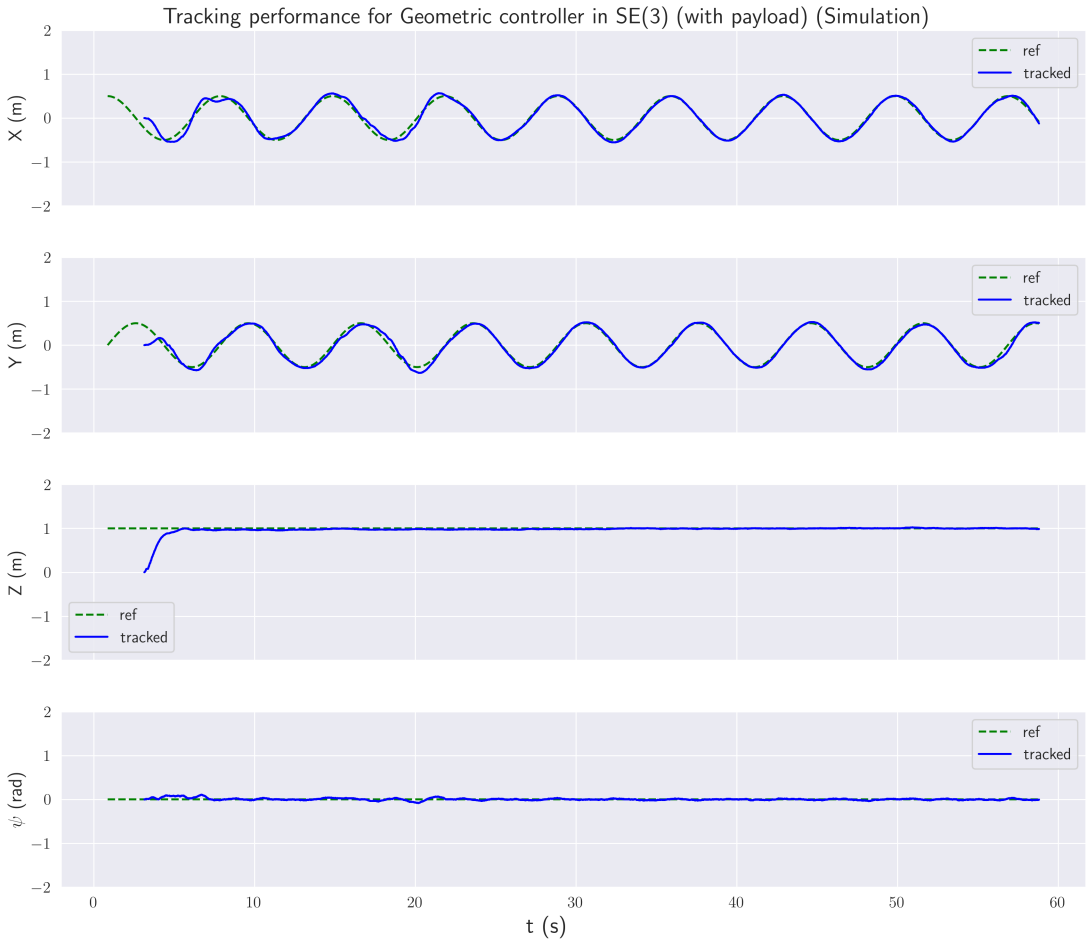

Figure 3.27: Tracking performance of UAV with payload in x-y-z-ψ with geometric controller in simulation

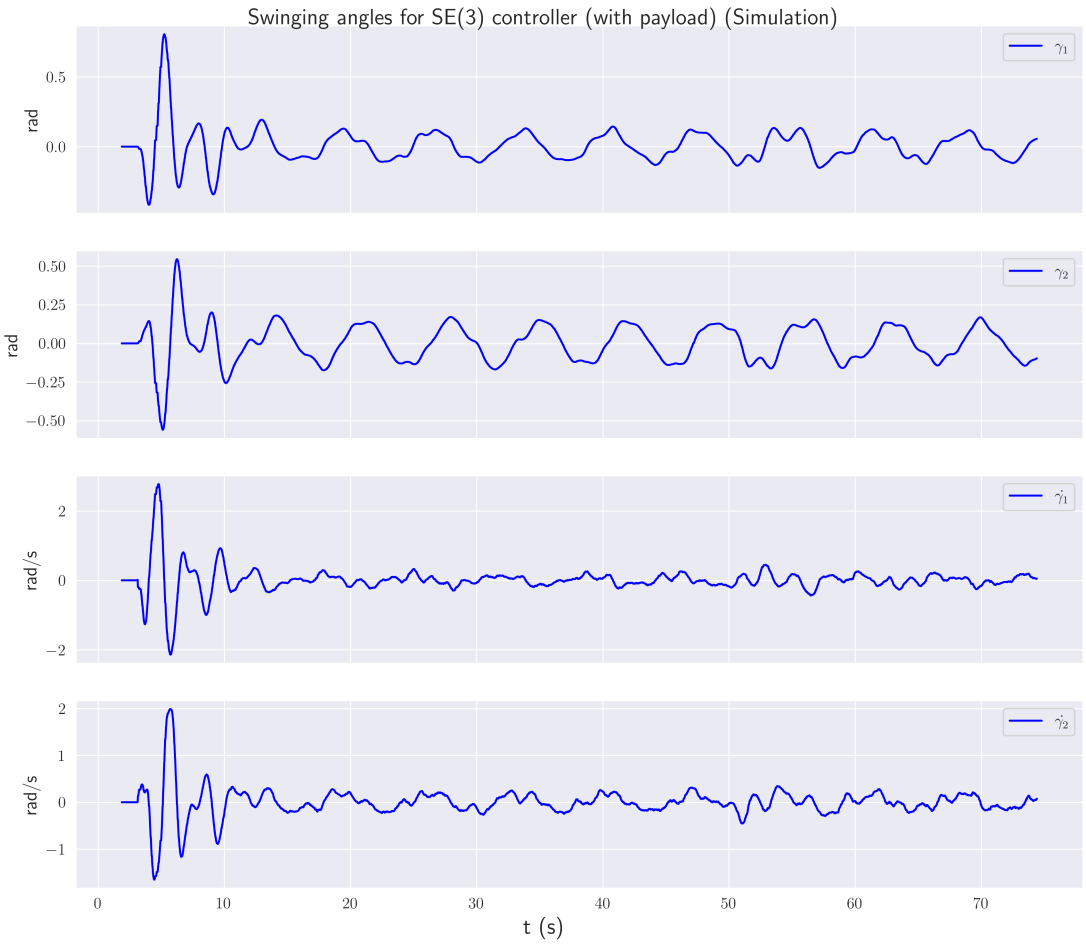

Figure 3.28: Swinging angles of payload with geometric controller in simulation

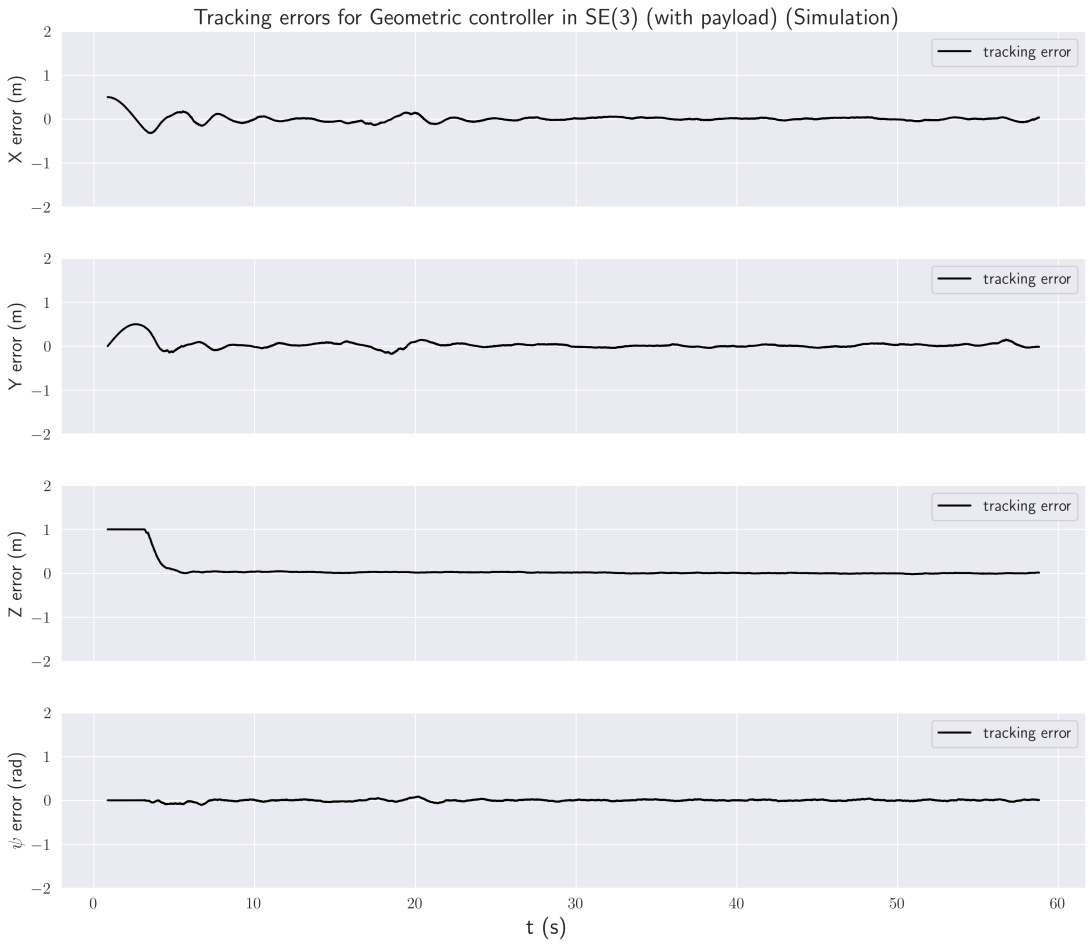

Figure 3.29: Tracking errors of UAV with payload in x-y-z-ψ with geometric controller in simulation

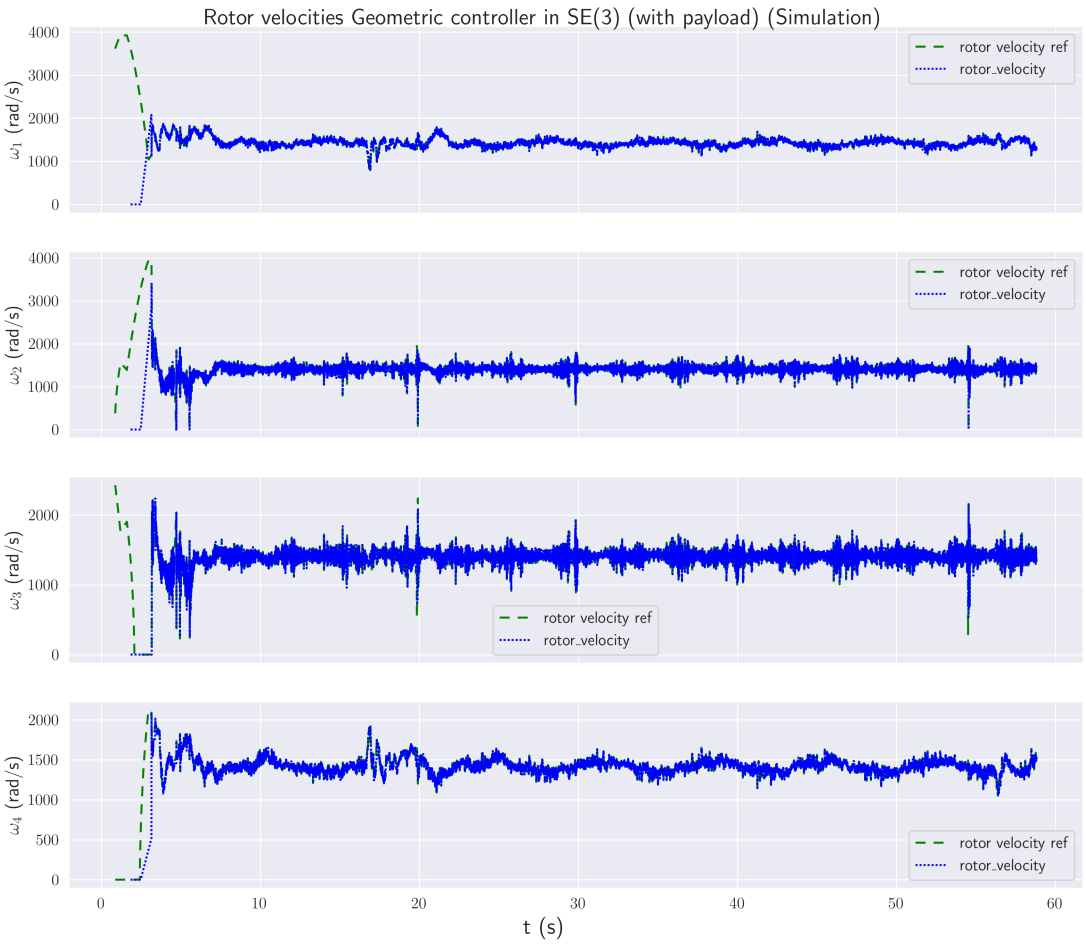

Figure 3.30: Commanded rotor velocities and the achieved rotor velocities with the geometric controller in simulation for UAV with payload

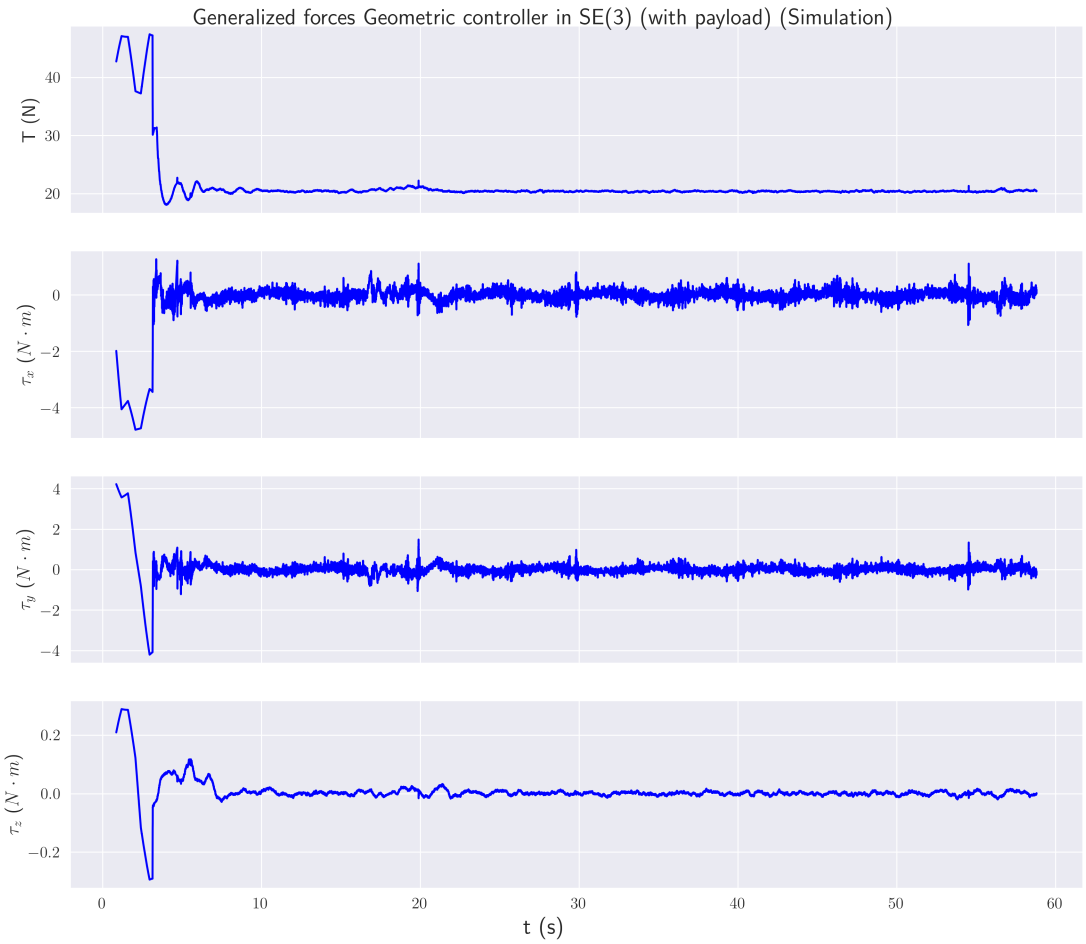

Figure 3.31: Virtual controls, the generalized forces, computed with the geometric controller in simulation for tracking control of UAV with payload

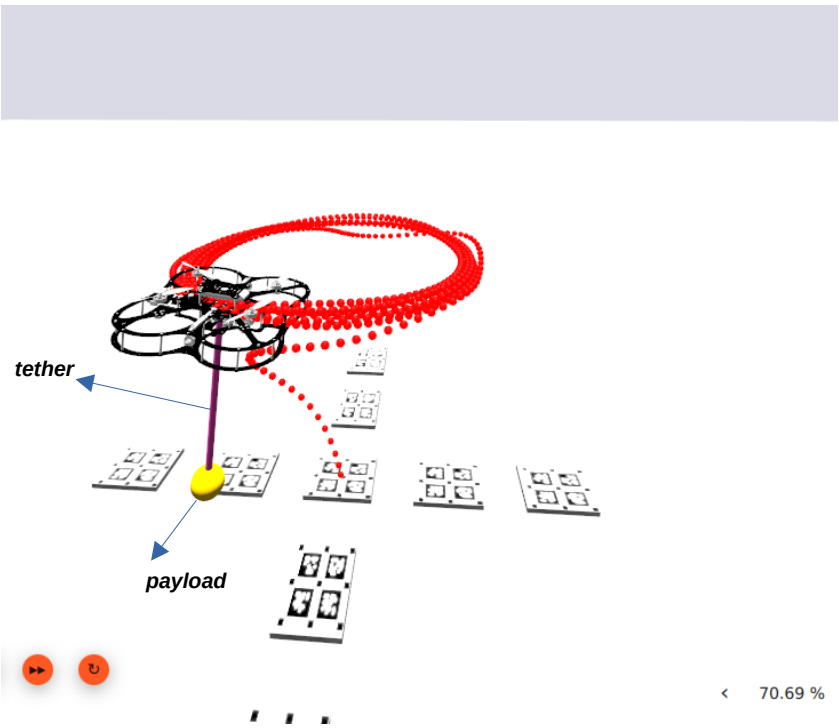

Figure 3.32: UAV with slung payload in Gazebo

Table 3.9: Tracking RMSE for x-y-z-ψ with geometric controller with payload

State	RMSE
x	0.036 m
y	0.042 m
z	0.053 m
ψ	0.011 rad

3.6.4 Comparison of PID, NMPC, and SE(3) in Simulation

The comparison between PID, NMPC, and SE(3) controllers (with and without payload) is shown below.

State	PID	NMPC	SE(3) w/o Payload	SE(3) w/ Payload
x, m	0.1213	0.080	0.1594	0.036
y, m	0.0385	0.060	0.14099	0.042
z, m	0.0377	0.0355	0.0835	0.053
ψ, rad	0.1179	0.012	0.0347	0.011

Table 3.10: RMSE comparison of PID, NMPC, and SE(3) controllers (with and without payload) on simulation

3.7 Comparison of Control Library

A comparison between our control library and the SOTA simulators used in literature is shown in Table 3.11[1][2].

Table 3.11: Comparison between our control library and other commonly used simulated environments for UAV

	Physics	**Renderer**	**Drone Model**			**Runtime Op.**		**Sync. / Steppable**	**Sensor**	**Task Spec.**
Env.	**Engine**		Quad.	Hexa.	Omni.	Conf.	Rand.	Phys.&Render.		
RotorS [31]	old Gazebo-based	OpenGL	✓	✓	✓	✗	✗	✗	*IMU, RGBD*	–
Airsim [11]	PhysX	Unreal Engine	✓	✗	✗	✗	✗	✗	*IMU, RGBD, S*	C&Python
Flightmare [30]	Flexible	Unity	✗	✓	✗	✗	✓	✓	*IMU, RGBD, S*	C++
PyBullet-Drones [58]	Bullet	OpenGL	✓	✗	✗	✗	✓	✓	*IMU, RGBD, S*	Python
FlightGoggles [90]	Flexible	Unity	✓	✓	✗	✗	✗	✗	*IMU, RGBD, S*	C++
CrazyS [91]	old Gazebo-based	OpenGL	✓	✓	✗	✗	✗	✗	*IMU, RGBD*	–
OmniDrones [59]	PhysX	Omniverse RTX	✓	✓	✓	✓	✓	✓	*IMU, RGBD, S, F, C*	Python
Ours	new Gazebo-based	OpenGL	✓	✓	✓	✓	✓	✓	*IMU, RGBD, S, F, C, T, SO, M, A*	C++&Python

[1]In Drone Model columns, Quad., Hexa., Omni. stand for quadcopter, hexacopter, and omnidirectional.

[2]In "Sensor" column, S = segmentation, F = force, C = contact, T = thermal, SO = sonar, M = magnetometer, A = air pressure

3.8 Evaluation of the Control System Toolbox

The proposed control system toolbox is evaluated from both a quantitative performance perspective with regards to accuracy, robustness, and computational feasibility and a qualitative capability perspective with regrads to modularity, integration flexibility, and breadth of supported UAV configurations. Quantitative metrics are drawn from the controller benchmarks in Section 3.6, while qualitative comparisons leverage the state-of-the-art (SOTA) analysis in Table 3.11.

3.8.1 Quantitative Evaluation

To verify the accuracy and robustness of the simulation environment, we implemented and tested a range of control strategies-PID, and NMPC in simulation and on hardware. Across all cases, the Root Mean Square Error (RMSE) in position and yaw tracking remained within $\mathcal{O}(10^{-1})$ m and $\mathcal{O}(10^{-2})$ rad, demonstrating high-fidelity simulation that closely matches hardware behavior.

Table 3.10 summarizes simulation RMSE for each controller, with and without payload. The close correspondence between simulation and hardware results in Tables 3.3, 3.8, 3.6, and 3.9 validates the dynamics modeling and disturbance injection (Ornstein–Uhlenbeck process) as an effective means of bridging the simulation-to-reality gap.

3.8.2 Qualitative Evaluation

From a capabilities standpoint, Table 3.11 highlights the distinctive features of the proposed toolbox relative to existing UAV simulation frameworks:

- Modularity and Language-Agnostic Design: Native support for both C++ and Python, enabling high-performance real-time controllers and rapid prototyping.

- Synchronization and Steppable Simulation: The `CustomServer` interface eliminates IPC timing overhead and enables on-demand stepping, which is critical for reinforcement learning and reproducibility.

- Sensor and Platform Diversity: A broader set of simulated sensors that includes IMU, RGBD, segmentation, force, contact, thermal, sonar, magnetometer, air pressure sensors, compared to frameworks.

The evaluation demonstrates that the toolbox not only achieves low tracking error across diverse controllers but also preserves fidelity when transitioning from simulation to hardware. Its modular architecture and synchronization capabilities enable use cases ranging from classical control validation to real-time reinforcement learning, surpassing the flexibility and integration options of current SOTA environments. The inclusion of disturbance models and slung-payload dynamics further extends the toolbox's applicability to safety-critical and complex operational scenarios.

4 GazeboDrones Reinforcement Learning Toolbox

4.1 Overview

Reinforcement learning is a subdomain of machine learning that has several applications in robotics. Researchers have applied RL to solve problems in navigation of autonomous robots while avoiding obstacles [92], robot control [93–95], autonomous driving [96, 97], and drone racing [56]. Deep Reinforcement Learning combines the representational power of neural networks with traditional RL, enabling it to solve complex problems with high-dimensional data [98].

Major innovations in RL occur through algorithmic improvements; however, other areas, such as improving sample efficiency and increasing the throughput of environments, are also significant [14]. High throughput in DRL can be achieved through large-scale computational clusters with advanced hardware tailored for AI, such as TPUs [99]. Academic research can be significantly accelerated by reducing the training time of Deep RL agents, enabling the fast prototyping of ideas and the tuning of reward functions. Specifically, in robotics training, RL agents that can be deployed in hardware can consume several hours, as demonstrated in [56]. As an example, the autonomous drone racing policy was trained for over 12 hours across 100 environments, collecting about 800 million samples.

Here, we specifically consider the parallelization aspect of environments to increase the throughput and simultaneously balance the model fidelity. As stated in the introduction, the Gazebo sim provides the perfect ecosystem for many robotic applications, making it a valid candidate for implementing environments for RL. However, the default packaging of Gazebo software is nowhere near ideal for the task, due to the IPC requirements, non-deterministic nature of execution, and limited parallelization capabilities. Furthermore, as stated in the last section, the default Gazebo software is not steppable. To solve the aforementioned problems, we introduced the `CustomServer`. Here, we extend the `CustomServer` to implement an OpenAI gym-compatible environment framework that can be highly parallelized in a single process with no IPC dependence, while guaranteeing experiments conducted in it can be reproduced anywhere.

Specifically, we leverage an Envpool [14] like execution style using thread-based concurrency that exploits multiple CPU cores to execute environments in parallel, and enhance our `CustomServer` based RL environment. Envpool addresses one of the major bottlenecks in RL systems, the interaction between the policy and the environment. For instance, in the IPC-based Gazebo environment, the policy and environment are interfaced through sockets, which causes nondeterminism due to the process scheduling of the operating system, leading to non-reproducible experiments. Furthermore, as the number of parallel environments grows, the IPC-based architectures will introduce significant delays due to the data marshalling and unmarshalling, a mechanism by which data from one process is serialized to be transported to another process and recovered. The Envpool execution style leverages Cython API (C Python) to execute several environments in parallel in optimized C++ code while allowing agents to obtain the observation from the environment through Python, all in the same process, while eliminating any overheads from IPC. This feature allows huge parallelization of our `CustomServer`, enabling researchers to train agents faster

and more efficiently, all with the high-fidelity UAV model simulated through Gazebo sim.

4.2 Preliminaries

The premise of Reinforcement learning starts with an Agent in an environment that satisfies the Markov property. The Agent is an actor that takes an action $a_t \in \mathcal{A}$ at each timestep based on an observation $O_t \in \mathcal{O}$ and receives a scalar reward $r_{t+1} \in \mathbb{R}$. RL is an optimization algorithm that aims to maximize the expected long-term reward modeled as the return.

$$G_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + ... = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \tag{4.1}$$

Since the environment is Markovian, the optimization problem of finding the optimal action given observation can be solved using Markov Decision Process, a framework for solving sequential decision-making problems [100]. Specifically, to satisfy the Markov property, the environment must satisfy the following conditions with $s_t \in \mathcal{S}$ defined as the state of the environment

$$p(s_{t+1}|s_t,\ s_{t-1},\ ...,\ s_0) = p(s_{t+1}|s_t)$$

$$O_t = s_t \tag{4.2}$$

Eq. (4.2) is formulated under the assumption that $t \in \{0,\ 1,\ ...,\ N\}$, that the trajectory of states is achieved in discrete steps, and is finite. This represents the condition that the next state of the environment is only conditioned on the previous state, and that the current state is a representative of all prior states. Eq. (4.2) represents the full observability condition, that is, the agent can observe the full state of the environment at each timestep. In real-life problems, Eq. (4.2) cannot be satisfied due to limitations on the hardware or the nature of the problem itself. In such cases, optimization requires leveraging variants of MDP, such as partially observable MDP [101].

For the remainder of this thesis, we will consider the fully observable MDP, unless specified. Hence, we will interchangeably use $O_t = s_t$.

4.2.1 MDP

To solve a decision-making problem under the MDP framework, we need to represent it using a quintuple $(\mathcal{S},\ \mathcal{A},\ \rho,\ f,\ \gamma)$. The $\mathcal{S}$ is the set of all possible states of the environment that will be considered in the problem, $\mathcal{A}$ is the set of all actions the agent can take that is conditionally distributed under $\mathcal{S}$ as a policy $\pi(a_t|s_t) \mid \pi : \mathcal{S} \to \mathcal{P}(\mathcal{A})$, $\rho : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$ is the reward function, $f : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$ is the state transition function modeled as a probability distribution under $\mathcal{S} \times \mathcal{A} \times \mathcal{S}$, and γ is the discount factor for weighting future rewards.

The goal of MDP is to determine an optimal sequence of actions under a policy π for the set of all possible finite trajectories (Fig. 4.1), τ, starting from a given state S_t that maximizes the return (Eq. (4.1)). Referring to Fig. 4.1, a trajectory $\tau_i \in \tau$ is a path from the root node to a leaf node. Since the state transition and the policy can be considered stochastic in the general form, the return in Eq. (4.1) is reformulated as the expectation over the $(\mathcal{S},\ \mathcal{A}) \sim \tau$, named as the state value function $V_\pi(s_t)$ for finite MDP [102]

$$
\begin{aligned}
V_\pi(s_t = S_t) &= \mathbb{E}_\pi\Big[G_t \mid s_t = S_t\Big] \\
&= \mathbb{E}_\pi\Big[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1}\Big] \\
&= \sum_{'a_t \sim \pi} \pi(' a_t | s_t = S_t) \sum_{' s} f(' s_t | s_t, ' a_t) \Big[r(' s_t, s_t, ' a_t) + \gamma \underbrace{\mathbb{E}_\pi\big[G_{t+1} \mid s_t = ' s_t\big]}_{V_\pi(' s)}\Big] \\
&= \sum_{'a_t \sim \pi} \pi(' a_t | s_t = S_t) \sum_{' s} f(' s_t | s_t, ' a_t) \Big[r(' s_t, s_t, ' a_t) + \gamma V_\pi(' s)\Big]
\end{aligned}
\tag{4.3}
$$

Here, we start with the initial state S_t and calculate the expectation of all possible

returns G_t by sampling actions a_t from policy π, and performing state transitions based on the $f(\mathcal{S} \mid \mathcal{S} \times \mathcal{A})$. The recursive Eq. (4.3) is known as the Bellman equation [103], which is a necessary condition for optimization of a problem under the dynamic programming paradigm.

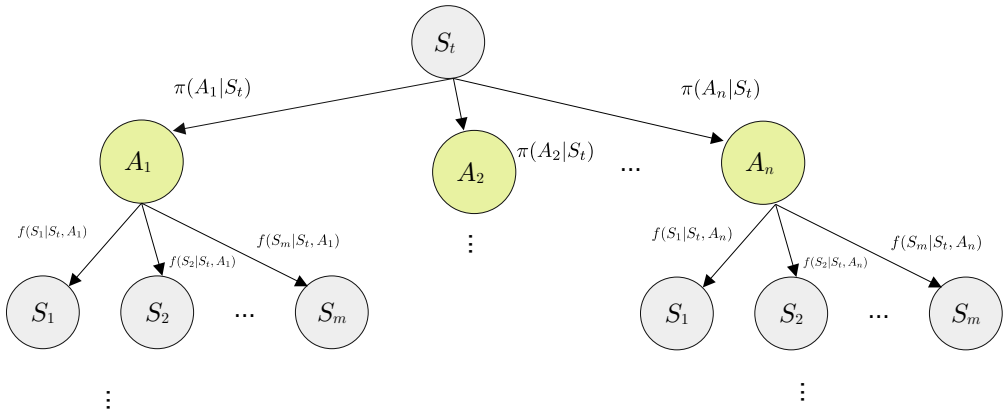

Figure 4.1: Illustration of Markov decision process under a policy π and state transition f

Similarly, we can also define the action value function, the value of taking a particular action a_t given a state S_t, thereby reducing the outermost expectation in Eq. (4.3) as

$$\begin{aligned} Q_\pi(s_t = S_t, a_t = A_t) &= \mathbb{E}_\pi\Big[G_t | s_t = S_t, a_t = A_t\Big] \\ &= \sum_{'s} f(' s_t | s_t, A_t)\Big[r(' s_t, s_t, A_t) + \gamma \underbrace{\mathbb{E}_\pi\big[G_{t+1} \mid s_t = {}' s_t\big]}_{V_\pi(' s)}\Big] \\ &= \sum_{'s} f(' s_t | s_t, A_t)\Big[r(' s_t, s_t, A_t) + \gamma V_\pi(' s)\Big]\Big] \end{aligned} \tag{4.4}$$

Note that

$$\begin{aligned} V_\pi(s_t) &= \mathbb{E}_{' a_t \sim \pi}\big[Q_\pi(s_t, {}' a_t)\big] \\ &= \sum_{' a_t} \pi(' a_t | s_t) Q_\pi(' a_t, s_t) \end{aligned} \tag{4.5}$$

4.2.2 Optimal Policy

The determination of a policy π_* that maximizes the return for a problem structured in the form of an MDP is the end goal of RL. The policy π_* is referred to as the optimal policy, which may not be unique, and must satisfy the following condition with $V_{\pi_*}(s_t)$ defined as the optimal state value function

$$V_{\pi_*} = \max_{\pi} V_{\pi}(s_t) \tag{4.6}$$

In other words, π_* is the policy that maximizes the state value function. Similarly, the π that maximizes Q_π is same as the one forV_{π_*}.

$$Q_{\pi_*} = \max_{\pi} Q_{\pi}(s_t) \tag{4.7}$$

Under Bellman optimality equation, $V_{\pi_*}(s_t) = \max_{a_t \in \mathcal{A}} Q_{\pi_*}(s_t, a_t)$, that is the action value for best action in a state under the optimal policy equals to the state value. Based on Eq. (4.3) and Eq. (4.4), this can be true if

$$\pi_*(a_t|s_t) = \begin{cases} 1 & \text{if } a_t = a_{*t} \\ 0 & \text{else} \end{cases} \tag{4.8}$$

$\pi(a_t = a_{*t}|s_t = S_t) = 1$ here, a_{*t} is the optimal action for the state S_t, which reduces Eq. (4.3) and Eq. (4.4) to

$$\begin{aligned} V_{\pi_*}(s_t = S_t) &= \sum_{'a_t \sim \pi_*} \pi(' a_t|s_t = S_t) \sum_{'s} f(' s_t|s_t, ' a_t) \Big[r(' s_t, s_t, ' a_t) + \gamma V_{\pi_*}(' s) \Big] \Big] \\ &= \sum_{'s} f(' s_t|s_t, a_{*t}) \Big[r(' s_t, s_t, a_{*t}) + \gamma V_{\pi_*}(' s) \Big] \Big] \\ Q_{\pi_*}(s_t = S_t, a_t = a_{*t}) &= \sum_{'s} f(' s_t|s_t, a_{*t}) \Big[r(' s_t, s_t, a_{*t}) + \gamma V_{\pi_*}(' s) \Big] \Big] \\ V_{\pi_*}(s_t = S_t) &= Q_{\pi_*}(s_t = S_t, a_t = a_{*t}) \end{aligned} \tag{4.9}$$

4.2.3 RL Algorithms

The methods to obtain the optimal policy in RL are referred to as the algorithms. This ranges from dynamic programming and Monte-Carlo methods to Policy gradient and Actor-Critic methods. In dynamic programming [104], the optimal policy is found by solving the Bellman optimality equations with a known environment model, whereas in Monte-Carlo methods, the state value is approximated using trajectory samples collected from the environment when the explicit modeling is not feasible. Temporal Difference (TD) [105] is another class of algorithms that is agnostic of the state transition function f. Precisely, TD bootstraps the value function with the update equation

$$\begin{aligned} V(s_t) &= V(s_t) + \alpha(V_{target}(s_t) - V(s_t)) \\ &= V(s_t) + \alpha(r_{t+1} + \gamma V(s_{t+1}) - V(s_t)) \end{aligned} \tag{4.10}$$

Here, $r_{t+1} + \gamma V(s_{t+1})$ is the TD target. A major advantage of TD methods is that they do not require a full trajectory sequence to learn, since the update equation only requires a single transition sequence $(s_t, a_t, s_{t+1}, r_{t+1}, a_{t+1})$.

The aforementioned algorithms require the estimation of optimal $Q_\pi(s_t)$ or $V_\pi(s_t)$, hence referred as value-based methods. Policy gradient algorithms are distinct from the aforementioned algorithms in the sense that the optimal policy is found by directly optimizing a parametrized π_θ based on an objective function, $\mathcal{J}$, that depends on the V_π. Precisely, the objective is defined as the performance measure of the policy π_θ, which is the value of the state.

$$\begin{aligned} \mathcal{J} &= V_{\pi_\theta}(s_t) \\ &= \sum_{'a_t \sim \pi_\theta} \pi_\theta(' a_t | s_t) Q_{\pi_\theta}(' a_t, s_t) \end{aligned} \tag{4.11}$$

The optimal policy $\pi_{*\theta}$ is found by performing gradient ascent on the parameters θ by

maximizing $\mathcal{J}$ [106].

$$\nabla_\theta \mathcal{J} \propto \sum_{s_t} \mu_{\pi_\theta}(s_t) \sum_{'a_t} \nabla_\theta \pi_\theta(' a_t|s_t) Q_{\pi_\theta}(' a_t, s_t) \tag{4.12}$$

Here $\mu_{\pi_\theta}(s_t)$ is the on-policy distribution of states, which is the weight assigned to each state based on the visitation frequency. Since $\mu_{\pi_\theta}(s_t)$ depends on the current policy π_θ, policy gradient is referred to as an on-policy method. This is because μ needs to be estimated based on the trajectories collected using π_θ. The parameters of the policy, θ, can be updated as

$$\theta_{t+1} = \theta_t + \alpha \nabla_\theta \mathcal{J} \tag{4.13}$$

Here α is the step size. The REINFORCE [102] is a policy gradient variant that uses a Monte-Carlo approach to estimate the state visitation frequency based on a set of sample trajectories, τ_s. Consider a set of trajectories collected with policy π_θ, $\tau_s = \{s_i^1, a_i^1, .., s_M^N, a_M^N\}$. Then, the averaging the inner expectation in Eq. (4.12) provides an approximation to the true policy gradient since the frequency of each state, in the sampled on-policy trajectory, is an estimation of the true visitation frequency. Therefore, Eq. (4.12) can be rewritten as follows

$$\begin{aligned}
\nabla_\theta \mathcal{J} &= \sum_{s_t} \mu_{\pi_\theta}(s_t) \sum_{'a_t} \nabla_\theta \pi_\theta(' a_t|s_t) Q_{\pi_\theta}(' a_t, s_t) \\
&= \mathbb{E}_{\pi_\theta}\Big[\sum_{'a_t} \nabla_\theta \pi_\theta(' a_t|S_t) Q_{\pi_\theta}(' a_t, S_t) \Big] \\
&= \mathop{\mathbb{E}}_{\pi_\theta}\Big[\sum_{'a_t} \pi_\theta(' a_t|S_t) \frac{\nabla_\theta \pi_\theta(' a_t|S_t)}{\pi_\theta(' a_t|S_t)} Q_{\pi_\theta}(' a_t, S_t) \Big] \\
&= \mathop{\mathbb{E}}_{\pi_\theta}\Big[\frac{\nabla_\theta \pi_\theta(A_t|S_t)}{\pi_\theta(A_t|S_t)} Q_{\pi_\theta}(A_t, S_t) \Big]
\end{aligned} \tag{4.14}$$

Here, we used the realization of the random variables to make an estimate of the quantities based on the Monte-Carlo approach, where $S_t \in \tau_s$ is the states visited using the policy, and $A_t \in \tau_s$ is the actions sampled from the policy distribution. The main idea

of this approach is to make an estimate of expectations involving $\pi_\theta(a_t|s_t)$ and $\mu(s_t)$ by averaging the samples.

Finally, Actor-Critic methods are a class of algorithms that learn parallel policy and value functions [102]. In this approach, the policy is used to generate state transitions from the environment, and the value function provides a metric to evaluate the policy. The policy is updated based on the policy gradient theorem.

4.2.4 DRL

Deep reinforcement learning is a powerful tool to solve RL optimization in the framework of machine learning. Traditionally, RL suffered from the curse of dimensionality due to the computational complexity arising from high-dimensional state spaces. With growing advancements in Neural networks and deep learning, the powerful representation capabilities of neural networks can be combined with RL to solve complex problems across computer vision and control domains.

The breakthrough in this field started with the Deep Q Network (DQN) [107] that extended the TD learning with neural networks. Traditionally, TD learning methods are considered to be unstable when leveraging non-linear function approximators, such as a neural network, to approximate the value function. This is greatly affected by the temporal correlation between training samples used for the TD update. DQN proposed using an experience replay buffer, disconnecting the samples temporally through uniform sampling, to prevent temporal correlation. Other value-based algorithms include Double DQN [108], Deep Recurrent Q-Network (DRQN) [109], and Difference maximization Q-Learning (DMQ) [110].

In this thesis, we will specifically consider a class of Policy-Based DRL algorithms that extend the policy gradient theorem to deep neural networks. Specifically, we consider

the Trust Region-Based algorithms that include Trust Region Policy Optimization (TRPO) [51], and Proximal Policy Optimization (PPO) [50]. The policy update based on Eq. (4.13) is highly sensitive to the step size α, requiring careful tuning for a stable policy update while ensuring monotonic convergence. TRPO overcomes these limitations by imposing a trust region constraint on the policy optimization based on a Kullback-Leibler (KL) divergence measure between the updated policy and the old policy. The objective to maximize in TRPO is

$$\begin{aligned} &\underset{\theta}{\text{maximize}}\ \mathbb{E}_t\left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta'}(a_t s_t)} A_{\pi_{\theta'}}(a_t, s_t)\right] \\ &\text{subject to}\ \mathbb{E}_t\Big[\mathrm{KL}(\pi_{\theta'}||\pi_\theta)\Big] \leq \delta \end{aligned} \tag{4.15}$$

where $\theta^{'}$ is the parameters of the policy before update, θ is the current policy parameters, $A_{\pi_{\theta'}}(a_t, s_t) = Q_{\pi_{\theta'}}(s_t, a_t) - V_{\pi_{\theta'}}(s_t)$ is the advantage of the policy for a given (s_t, a_t), and δ is the upper bound for the soft constraint on KL divergence. For a given trajectory collected under some policy, the $\pi_{\theta'}(a_t, s_t)$ remains constant and is equivalent to the policy from which the data was collected. The term $\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta'}(a_t|s_t)}$ is referred to as the importance weight, which arises from the application of importance sampling, a method to compute expectation of a random variable under probability density, π_θ for instance, under another probability density, $\pi_{\theta'}$ for instance.

In PPO, the TRPO objective is modified to introduce a set of algorithms that improve the training process. Specifically, PPO includes KL penalty and a clipped penalty algorithm derived from the TRPO objective. In the KL penalty objective, the soft constraint is included in the objective, converting the optimization into an unconstrained optimization.

Specifically, the KL penalty PPO optimization problem is defined as

$$\underset{\theta}{\text{maximize}}\ \mathbb{E}_t\left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta'}(a_t|s_t)} A_{\pi_{\theta'}}(a_t, s_t) - \beta \mathrm{KL}(\pi_{\theta'}||\pi_\theta)\right] \tag{4.16}$$

A major practical challenge is tuning the parameter β, which often requires adaptation as the policy evolves during training [50]. Furthermore, KL divergence explosions during training can cause instability.

Clipped PPO objective introduces a surrogate objective, which is defined as follows

$$
\begin{aligned}
r_t(\theta) &= \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta'}(a_t|s_t)} A_{\pi_{\theta'}}(a_t, s_t) \\
\underset{\theta}{\text{maximize}}\ \mathbb{E}_t &\Big[\min(r_t(\theta) A_{\pi_{\theta'}}(a_t, s_t),\ \text{clip}(r_t(\theta),\ 1-\epsilon,\ 1+\epsilon) A_{\pi_{\theta'}}(a_t, s_t)) \Big]
\end{aligned}
\tag{4.17}
$$

Here $r_t(\theta)$ is clipped between $(1-\epsilon,\ 1+\epsilon)$ for some hyperparameter $\epsilon < 1$.

4.3 DRL Library

Leveraging the Steppable simulation framework introduced in Section. 3.3, we present the GazeboDrones Deep Reinforcement Learning (DRL) toolbox—a framework designed to support deep learning research with UAVs. By combining high-fidelity UAV dynamics in the Gazebo ecosystem with a parallelizable DRL training infrastructure, this toolbox enables efficient training and data collection for robotics applications.

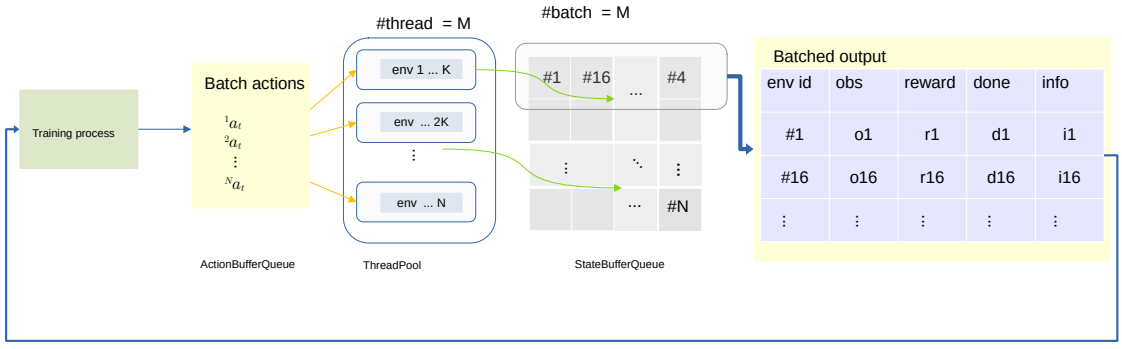

Figure 4.2: Overview of the GazeboDrones DRL toolbox

An overview of the system is shown in Fig. 4.2. The design supports asynchronous, event-driven execution, enabling high sample rates and efficient experience collection—two

common bottlenecks in DRL for robotics. A multi-threaded executor, `ThreadPool` [111], is used to parallelize multiple environments, while a queue-based communication system handles action dispatch and observation collection. Specifically, incoming actions are pushed into the `ActionBufferQueue` and delivered to their corresponding environments in FIFO (First Come, First Out) order. Similarly, observations are stored in the `StateBufferQueue`, converted into a `NumPy` [112] array, and forwarded to the Python front-end once a specified batch size is reached.

The key features of the toolbox are:

- A high-fidelity, customizable environment framework for DRL and Embodied AI applications, built around UAVs simulated in the Gazebo ecosystem
- Scalable training using `torchrl` [113], supporting efficient hardware utilization
- Flexible simulation control via the `CustomServer`, supporting both deterministic and non-deterministic experiments
- Policy export via TensorRT [114], facilitating rapid transfer from simulation to hardware

4.3.1 DRL Environment

To facilitate accelerated prototyping of UAV DRL applications, our library extends the `CustomServer` to `DRLServer`, a variant that forms the bedrock for customized environments. Specifically, the `DRLServer` implements a set of Application Programming Interface (API) (Fig. 4.3) that enables downstream environments to customize the agent observation space and action space, while implementing an Envpool style execution (4.2), enabling massive parallelization. This allows researchers to create complex environments through SDF and obtain the world state information that includes the sensor states through our

`DRLServer`. Precisely, the downstream code can choose from a subset of the world state as the observation. Furthermore, the action space specification can be categorized into high-level global or relative state commands and low-level virtual controls or actuator commands. The high-level action commands are tracked by a low-level controller using our Control toolbox, which can be specified by the downstream user.

To further enhance rapid prototyping of DRL applications, we define another set of specifications following the OpenAI gym API, which includes the interface methods `Step` and `Reset` as well as the specification of observation and action space dimensions. Specifically, custom environments can inherit our `DRLServer` and leverage its API to define an MDP-compatible environment and agent, and perform efficient training using our toolbox. A fully defined custom environment that has implemented the interface methods can then be compiled into an optimized CPython library using our DRL toolbox and utilized from Python training code, enabling researchers to leverage popular deep learning libraries like PyTorch while maintaining fast environment execution.

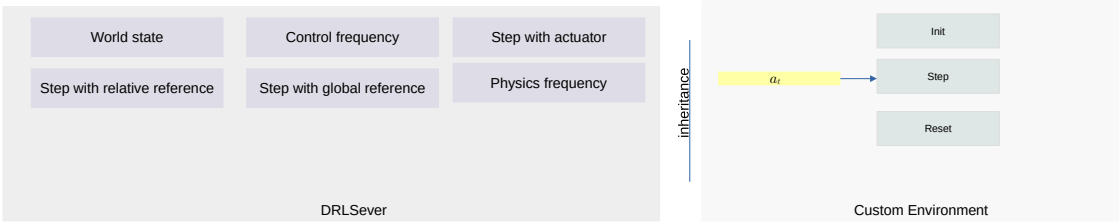

Figure 4.3: Illustration of our DRLServer API

4.3.1.1 DRLServer

`DRLServer` extends the capabilities of `CustomServer` by incorporating functionalities specifically designed for Deep Reinforcement Learning (DRL) environments. Beyond the features already offered by `CustomServer`, `DRLServer` enables direct access to simulation sensor

data, including RGBD images, point clouds, IMU readings, and more. It is also seamlessly integrated with the Envpool architecture (see Fig. 4.2), facilitating the deployment of scalable and high-performance simulation environments. Additionally, a clearly defined API supports modular customization, allowing `DRLServer` to be adapted to application-specific requirements.

4.3.1.2 DRLServer API

Many widely used reinforcement learning (RL) libraries rely on a standardized set of environment interfaces, most notably those defined by the OpenAI Gym API. This includes the `step` and `reset` methods, as well as definitions for the action and observation spaces. The `step` function typically takes an action a_t as input and returns the next state s_{t+1}. The `reset` function is used to reinitialize the environment following terminal conditions, such as collisions or task completion. Specifications of the action and observation space dimensions allow RL libraries to automatically configure the appropriate input and output shapes for policy networks.

To support this standardized API, the simulation environment exposes world state data via the `DRLServer`, and supports agent state propagation through three distinct methods: *Step with relative reference*, *Step with global reference*, and *Step with actuator*. The relative and global reference modes update the simulation state using a configured controller, implemented via our Control library. In the relative mode, a global target is derived from the current state and a provided relative reference. In contrast, the actuator mode directly updates the agent state by applying rotor velocities.

This modular approach enables policy training at varying levels of abstraction, from high-level navigation to low-level motor control.

For environment resets, the system allows arbitrary placement of agents and other sim-

ulation entities. This feature facilitates the implementation of advanced training strategies, such as *curriculum learning*, where task difficulty progressively increases in response to the agent's learning progress.

4.3.2 Training Platform

To further enable efficient research, we develop a training platform leveraging the `torchrl` library that can automatically execute parallel environments and train DRL agents, exploiting our DRL toolbox. Specifically, we integrate our Envpool-based environments directly into the `torhcrl` framework, enabling researchers to conduct experiments using diverse DRL algorithms. Furthermore, our platform provides a feature to export the trained policies to an `ONNX` [115] runtime and optionally a CUDA-engine based inference executable using TensorRT, enabling direct deployment in hardware.

4.4 Drone Racing with PPO

First-person view (FPV) drone racing is a professional sport with the goal of piloting a UAV through a circuit. Recent works such as [56] have demonstrated the feasibility of deploying an RL agent trained entirely on simulation to compete against professional pilots. This was a remarkable feat due to the difficulties inherent in the problem arising from the need to coordinate several sensory inputs while performing precise trajectory planning.

Typically, the AI-based system consists of a perception subsystem and a control subsystem. The former performs state estimation using techniques such as multi-sensor fusion, Visual Inertial Odometry (VIO), etc., to provide feedback, whereas the control system is concerned with tracking the higher-level reference trajectory. In particular, [56] leveraged a Kalman Filter to fuse partial state estimates from IMU, VIO, and raw Grayscale Image, and utilized a learned PPO policy as the controller. Specifically, the policy was

a parametrized Multilayer Perceptron (MLP) with two hidden layers of dimensions 128 that was inferred at 100 Hz, and was used as an outer-loop controller with the actions $\begin{bmatrix} T_{ref} & {}^{B}\omega_{x,\,ref} & {}^{B}\omega_{y,\,ref} & {}^{B}\omega_{z,\,ref} \end{bmatrix}$

Here, we leverage our DRL toolbox to learn a policy that performs higher-level trajectory planning similar to the work [57]. In particular, we attempt to train a policy that can plan a time-optimal trajectory to pass through a predefined sequence of gates [116], with the observation space s_t defined as

$$s_t \in \mathbb{R}^{45} := \begin{bmatrix} {}^{I}p & {}^{I}v & {}^{B}_{I}R & {}^{B}\omega & a_{t-1} & {}^{I}\delta p_1 & {}^{I}\delta p_2 \end{bmatrix} \tag{4.18}$$

where ${}^{I}p$ is the inertial frame position, ${}^{I}v$ is the inertial frame velocity, ${}^{B}_{I}R$ is the rotation matrix from inertial frame to body frame, ${}^{B}\omega$ is the angular velocity of the body frame, a_{t-1} is the previous action from the policy, and ${}^{I}\delta p_i \in \mathbb{R}^{12}$ is the relative position to next i^{th} gate corners. The action space is defined as

$$a_t := \begin{bmatrix} {}^{I}\delta x & {}^{I}\delta y & {}^{I}\delta z \end{bmatrix} \tag{4.19}$$

The action space is the relative position command. Precisely, for a given a_t, the low-level controller will track ${}^{I}p + a_t$. We use a control frequency of 100 Hz and a physics frequency of 1000 Hz, specified using our `DRLServer`. We also define a custom circuit for the racing with the configuration shown in Fig. 4.4

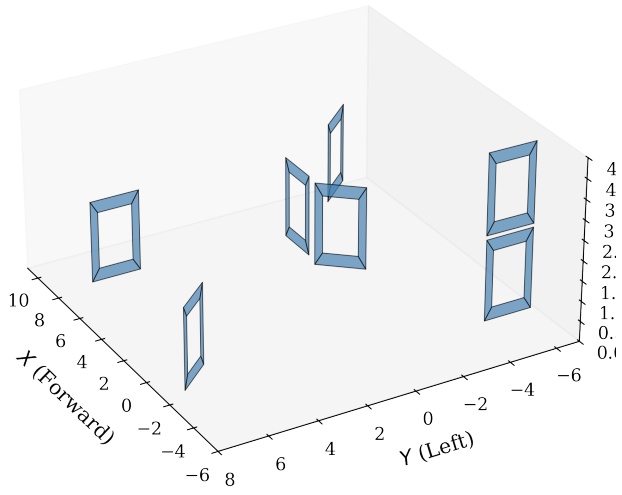

Figure 4.4: Racing circuit

The reward function is defined as

$$\begin{aligned} r_t &= r_{prog} + r_{act} + r_{angular} + r_{crash} \\ &= \lambda_1(d_{t-1} - d_t) + \lambda_2||a_t - a_{t-1}||^2 + \lambda_3||^B\omega|| + r_{crash} \end{aligned} \tag{4.20}$$

Here, r_{prog} is the reward for making progress along the circuit, defined as the difference between the distance to the next gate, d_t, at the last timestep and the current timestep, r_{act} is the term to encourage smooth action commands, $r_{angular}$ is a term to discourage high angular motion, and r_{crash} is a penalty term to discourage undesirable states such as a flipped down drone or collision with other entities. Specifically,

$$r_{crash} = \begin{cases} -5 & \text{if collided or flipped} \\ 0 & \text{else} \end{cases} \tag{4.21}$$

We use the following hyperparameters

$$\begin{aligned} \lambda_1 &= 1.0 \\ \lambda_2 &= -1e-4 \\ \lambda_3 &= -2e-4 \end{aligned} \tag{4.22}$$

We leverage our `DRLServer` API to obtain the desired state information, as well as execute the actions from the policy. Furthermore, we leverage our Geometric controller introduced in the Control library to perform the low-level control. The racing circuit realization in Gazebo is shown in Fig. 4.5

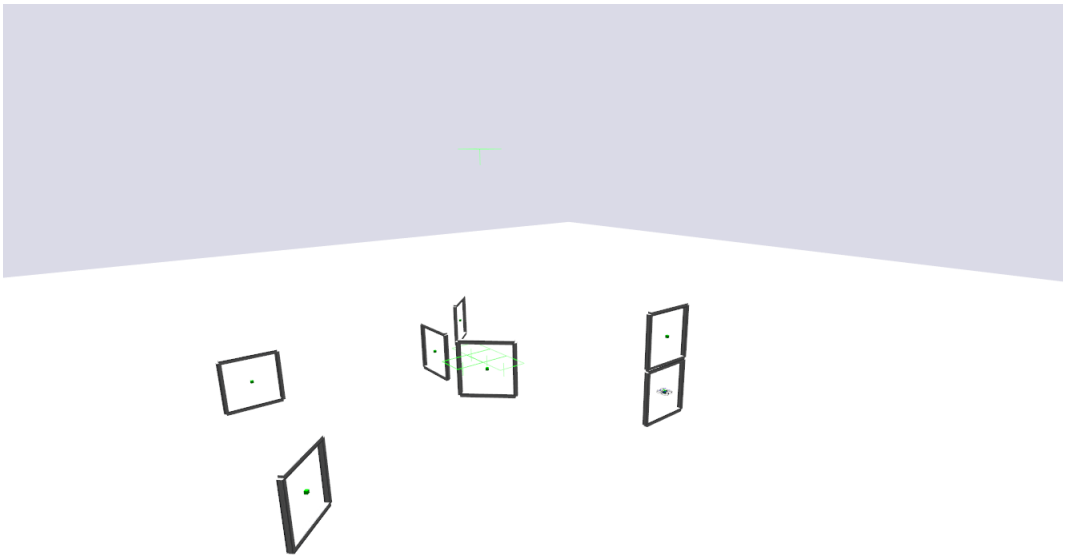

Figure 4.5: Racing circuit implemented in Gazebo

4.4.1 Training Process

We leverage our DRL training platform and the PPO algorithm to optimize an MLP policy with two hidden layers of size 256 and Leaky-ReLU activation with a negative slope of 0.2 (Fig. 4.6). The policy outputs the latent variables associated with the action

space, the mean μ, and standard deviation Σ. Specifically, we construct a Multivariate Gaussian Distribution using the latents from the policy. The samples from this distribution are then squashed using tanh function, and scaled by $\begin{bmatrix}1.0 & 1.0 & 1.5\end{bmatrix}$. In other words, the drone can be commanded to track a relative displacement of $\begin{bmatrix}|\delta x| & |\delta y| & |\delta z|\end{bmatrix} \leq \begin{bmatrix}1.0 & 1.0 & 1.5\end{bmatrix}$ at each step. We also leverage a critic network, with exact architecture as the policy, to estimate the value function V_π, which is used in the calculation of the advantage function. The advantage is computed using Generalized Advantage Estimation (GAE) and is provided by `torchrl`

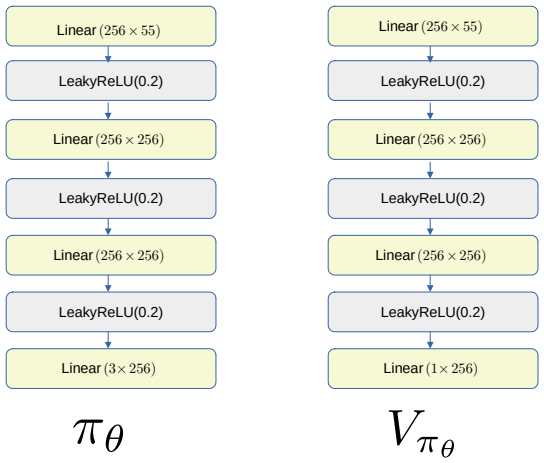

Figure 4.6: Policy and value function architecture

We initialize 100 racing environments in parallel on a workstation with 20 core Intel Xeon W-1290P CPU and 8 Gb NVIDIA Quadro RTX 4000 GPU and train for approximately 13 hours to converge, collecting a total of 24 million experiences, or equivalently 240 million physics steps, averaging about $5200\frac{steps}{sec}$.

The critic network is optimized based on a smooth L1 loss defined as

$$l_n = \begin{cases} \frac{0.5(x-y)^2}{\beta} & \text{if } |x-y| < \beta \\ |x-y| - 0.5\beta & \text{otherwise} \end{cases} \tag{4.23}$$

where we use $\beta = 1.0$.

Furthermore, we clip all the gradients so that their total norm is 1.0 to stabilize the training process. Each training epoch consists of on-policy data collection with a batch size of 102400. This batch is first used to estimate the advantage using the critic net and GAE, followed by randomization and splitting into minibatches of size 4096 to ensure training samples are i.i.d. (independent and identically distributed). This step is crucial to ensure there are no temporal correlations in the training data, which can cause unstable training. We then update the policy and critic networks with each minibatch using separate optimizers, with a learning rate of $3e-5$ for policy and $1e-4$ for the critic, until the policy has a KL divergence of 0.15 from the initial policy.

Important hyperparameters are summarized in Table 4.1.

4.4.2 Results

Here we show the results from the policy optimization

Figure 4.7: Episodic average reward during training

Figure 4.8: Clipped PPO loss over training

Figure 4.9: Critic loss over training

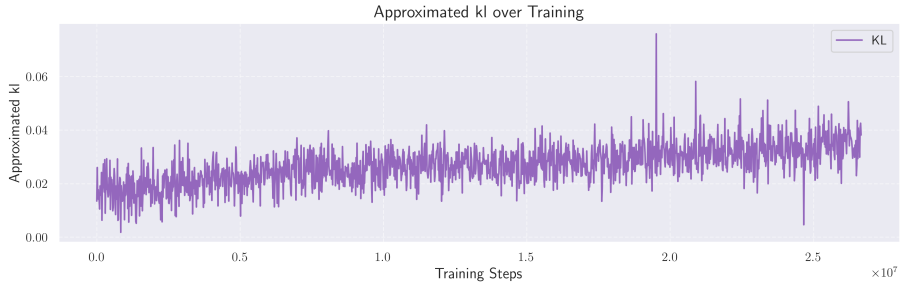

Figure 4.10: Approximated KL for each policy updates

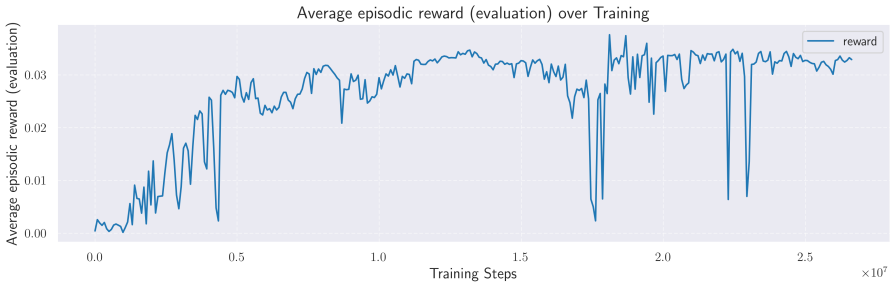

Figure 4.11: Evaluation reward (evaluated every 10 training epochs)

The tracking results are

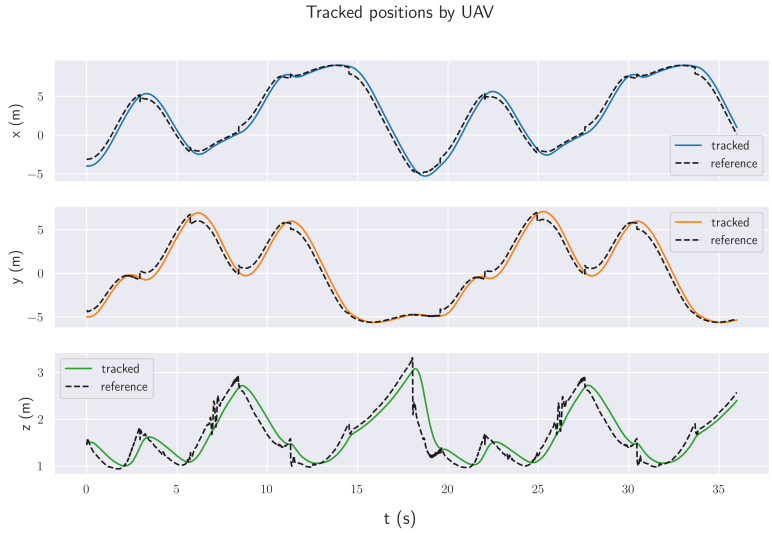

Figure 4.12: Tracking results, the reference is generated by adding the policy output with the current position

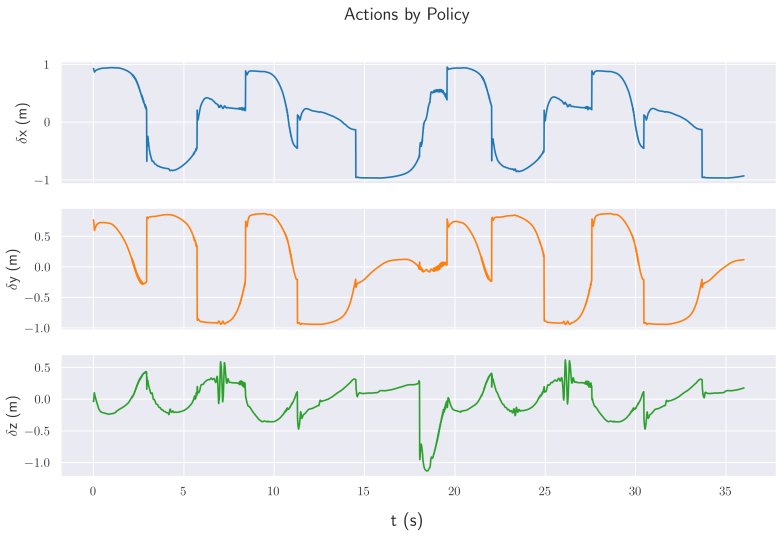

Figure 4.13: Policy actions (for Fig. 4.12)

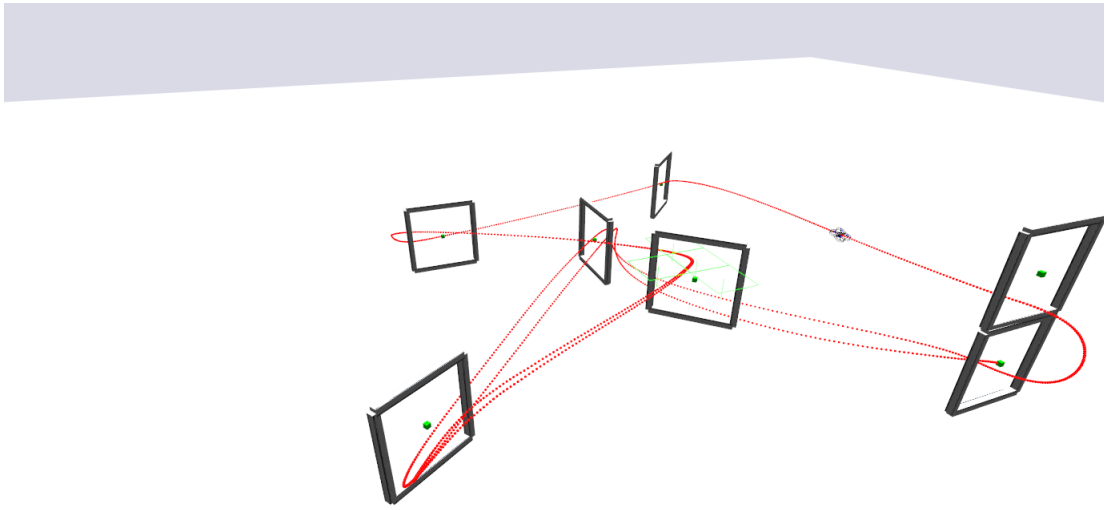

Figure 4.14: Tracked trajectory in Gazebo

4.5 Parallelization Performance

Here we report the scaling performance of environments on our workstation with 20 core Intel Xeon W-1290P CPU.

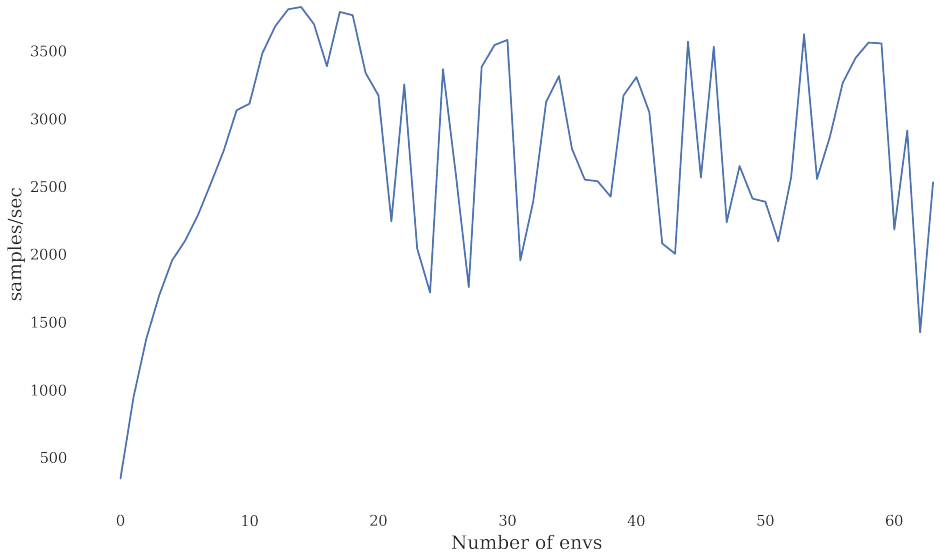

Figure 4.15: Experiences collected per second against the parallel environments on a 20 core CPU. Note that the actual number of physical samples generated is 10× since each experience requires 10 physics steps because the control frequency is 100 Hz with a physics frequency of 1000 Hz

4.6 Comparison of DRL Library

A comparison of our DRL library across other popular UAV platforms are shown in Table 4.2[1][2].

[1]In Drone Model columns, Quad., Hexa., Omni. stand for quadcopter, hexacopter, and omnidirectional.

[2]In "Sensor" column, S = segmentation, F = force, C = contact, T = thermal, SO = sonar, M = magnetometer, A = air pressure

4.7 Evaluation of DRL Toolbox

To assess the effectiveness of the GazeboDrones DRL toolbox, we evaluate it on two dimensions: performance, and flexibility. The goal is to quantify the benefits of the proposed environment design over other RL setups. We benchmark our DRL library quantitatively based on the quality of the simulation and flexibility.

To assess simulation quality, we measure both the physical fidelity and the computational performance of the environments. Physical fidelity is validated by comparing simulated UAV dynamics against real-world flight. This was achieved in the previous chapter. The average root-mean-square (RMS) tracking error between simulated and real-world flights was found to be less than 3.5 cm for hover and less than 6.8 cm for aggressive maneuvers, demonstrating close correspondence to real hardware.

On the computational side, we measure the throughput in terms of experiences per second as a function of the number of parallel environments. The results, shown in Fig. 4.15, confirm near-linear scaling up to 18 parallel environments before saturating CPU resources. This performance directly translates to reduced training times for DRL agents compared to conventional Gazebo-based RL setups.

Flexibility in the proposed DRL toolbox stems from the modular architecture of the simulation environment and the integrated rotor allocation algorithm. The modular design allows the separation of UAV dynamics, sensor models, and control allocation logic into independent components, enabling rapid reconfiguration without altering the simulation core. Our allocation algorithm supports arbitrary rotor layouts and propulsion system parameters, allowing the same DRL environment to be adapted to different vehicle geometries or failure scenarios by simply modifying configuration files. This design ensures that research on advanced control strategies, such as fault-tolerant control or morphing UAVs,

can be conducted without rewriting environment code.

These results demonstrate that the proposed DRL toolbox achieves high simulation fidelity, guarantees reproducibility, and provides substantial flexibility for rapid prototyping of UAV-based DRL research.

Table 4.1: Hyperparameter settings used for training

Hyperparameter	**Value**
Policy Learning Rate	3e-5
Critic Learning Rate	1e-4
Discount Factor (γ)	0.99
Batch Size	102400
Minibatch Size	4096
Optimizer	Adam
Policy Architecture	$\begin{bmatrix} 256 & 256 \end{bmatrix}$
Critic Architecture	$\begin{bmatrix} 256 & 256 \end{bmatrix}$
Number of Environments	100
Total Timesteps	24e6
Entropy Coefficient	1e-6
Value Loss Coefficient	1.0
Clip Range (PPO)	0.1
λ_{GAE}	0.95
λ_1	1
λ_2	-1e-4
λ_3	-2e-4
$\mathrm{KL}_{\max}$	0.15

Table 4.2: Comparison of popular UAV simulation platforms with deep reinforcement learning features

	Drone Models			**Runtime Op.**		**Sync. / Steppable**	**Vectorization**	**Sensors**	**User Interface**
Simulator	Quad.	Hexa.	Omni.	Conf.	Random				
AirSim	✓	✗	✗	✓	✗	✗	MP	*IMU, RGBD, S*	C++ & Python
Flightmare	✓	✗	✗	✓	✓	✓	MP	*IMU, RGBD, S*	C++
PyBullet-Drones	✓	✗	✗	✓	✓	✓	MT/MP	*IMU, RGBD, S*	Python
OmniDrones	✓	✓	✓	✓	✓	✓	MT/MP	*IMU, RGBD, S, F, C*	Python
Ours	✓	✓	✓	✓	✓	✓	MT/MP	*IMU, RGBD, S, F, C, SO, M, A*	C++&Python

5 Conclusion and Future Work

5.1 Conclusion

This thesis proposes a digital twin platform to accelerate autonomous research with UAVs, which is modular and scalable, enabling researchers to turn concepts into reality much faster. We first proposed a high-fidelity dynamics modeling based on combining the rotor blade element theory, aerodynamic forces, and rigid body dynamics, supplemented with an additive Gaussian-Markov-based dynamics. This modeling is coupled with an actuator-level dynamics model, relating the propeller rotation speed to the reference speed to create a highly precise digital twin model. Our model is then simulated in a Gazebo-based ecosystem, leveraging the tried-and-true framework for general robotics. Gazebo provides a reliable ecosystem for real-world deployment, further enhancing our digital twin platform, enabling researchers to realize concepts in our platform. Precisely, we develop a suite of `SystemPlugins` leveraging the Gazebo sim C++ API to simulate the control allocation algorithm based on our modeling, and bind them using CPython to expose their functionalities in Python.

The tightly-coupled architectures of existing simulators prevent them from being easily extendable to diverse applications, as well as limiting their interfacing capabilities. We address this through the introduction of `CustomServer`, a new approach to simulate with

Gazebo while exposing its functionalities in Python and C++ simultaneously, enabling researchers to integrate open-source code into their simulation seamlessly. Furthermore, our `CustomServer` facilitates massive same-process parallelization, further diversifying the use cases, such as Deep Reinforcement Learning. We provide a comparison of our simulator with other popular simulators in the modeling chapter.

To facilitate rapid prototyping and validation of control systems for UAVs, we introduced our Control Library, a suite that incorporates our high-fidelity model and Gazebo to provide convenient methods to design, validate, and deploy controllers. We validated our digital twin model by first implementing controllers ranging from PID to Model Predictive Controller using our control library and deploying them in the QDrone2 platform.

To address the needs of Deep Reinforcement Learning researchers, we extended our `CustomServer` with the DRL toolbox, a suite enabling accelerated research in DRL with our platform, with huge parallelization capabilities allowing researchers to prototype DRL experiments reliably and efficiently. We demonstrate the capabilities of our toolbox by training a PPO-based Autonomous Racing policy that performs time-optimal path-planning. The policy was trained over 240 million physics samples, over 13 hours of training, averaging about 5000 samples per second, collected from 100 parallel environments executed in a single process. We leveraged the Envpool-based execution style to enable massive in-process parallelization of environments to efficiently utilize hardware resources.

5.2 Future Work

This thesis presented a high-fidelity UAV model simulated in a Gazebo ecosystem. The main focus of this work was to develop a platform to accelerate research in UAV applications. Recent trends in robotics are shifting towards exploiting Large Language Models (LLMs) and Vision Language Models (VLMs) to solve problems in perception and planning.

While Gazebo provides a reliable simulation of dynamics, the rendering engine lacks many features required for applications involving VLMs. The Gazebo rendering engine can be improved by leveraging photorealistic engines such as Unity and Unreal Engine; however, this requires significant effort in integrating with the existing Gazebo architecture.

Our dynamic modeling can be improved further by incorporating learned dynamics, referred to as the residual model. This is typically implemented as a neural network that learns the residual model online, such as in [34]. This will be an interesting direction to pursue, to push the model fidelity even further, and to potentially enable direct simulation-to-hardware deployment of learned policies. Furthermore, another type of propeller dynamic model incorporating the BEM theory, which combines the blade element theory with the momentum theory.

References

[1] Quanser, "QDrone 2," https://www.quanser.com/products/qdrone-2/, 2025, accessed: 2025-05-25.

[2] Open Source Robotics Foundation, "Gazebo architecture," https://gazebosim.org/docs/latest/architecture/, 2024, accessed: 2025-05-29.

[3] G. Muchiri and S. Kimathi, "A review of applications and potential applications of uav," in *Proceedings of the Sustainable Research and Innovation Conference*, 2022, pp. 280–283.

[4] M. Ghamari, P. Rangel, M. Mehrubeoglu, G. S. Tewolde, and R. S. Sherratt, "Unmanned aerial vehicle communications for civil applications: A review," *IEEE Access*, vol. 10, pp. 102 492–102 531, 2022.

[5] M. Yuan, J. Shan, and K. Mi, "Deep reinforcement learning based game-theoretic decision-making for autonomous vehicles," *IEEE Robot. Autom. Lett.*, vol. 7, no. 2, pp. 818–825, 2021.

[6] ——, "From naturalistic traffic data to learning-based driving policy: A sim-to-real study," *IEEE Transactions on Vehicular Technology*, vol. 73, no. 1, pp. 245–257, 2023.

[7] M. Yuan, J. Shan, and H. Schofield, "Scalable game-theoretic decision-making for self-driving cars at unsignalized intersections," *IEEE Transactions on Industrial Electronics*, vol. 71, no. 6, pp. 5920–5930, 2023.

[8] T. Erez, Y. Tassa, and E. Todorov, "Simulation tools for model-based robotics: Comparison of bullet, havok, mujoco, ode and physx," in *2015 IEEE international conference on robotics and automation (ICRA)*. IEEE, 2015, pp. 4397–4404.

[9] M. A. Tahir, I. Mir, and T. U. Islam, "A review of uav platforms for autonomous applications: Comprehensive analysis and future directions," *IEEE Access*, vol. 11, pp. 52 540–52 554, 2023.

[10] A. Mairaj, A. I. Baba, and A. Y. Javaid, "Application specific drone simulators: Recent advances and challenges," *Simulation Modelling Practice and Theory*, vol. 94, pp. 100–117, 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1569190X19300048

[11] S. Shah, D. Dey, C. Lovett, and A. Kapoor, "Airsim: High-fidelity visual and physical simulation for autonomous vehicles," in *Field and Service Robotics: Results of the 11th International Conference*. Springer, 2018, pp. 621–635.

[12] "Gazebo - staging.gazebosim.org," https://staging.gazebosim.org/, [Accessed 01-02-2024].

[13] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," *arXiv preprint arXiv:1606.01540*, 2016.

[14] J. Weng, M. Lin, S. Huang, B. Liu, D. Makoviichuk, V. Makoviychuk, Z. Liu, Y. Song, T. Luo, Y. Jiang, Z. Xu, and S. Yan, "EnvPool: A highly parallel reinforcement learning environment execution engine," in *Advances in Neural*

Information Processing Systems, S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, Eds., vol. 35. Curran Associates, Inc., 2022, pp. 22 409–22 421. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2022/file/8caaf08e49ddbad6694fae067442ee21-Paper-Datasets_and_Benchmarks.pdf

[15] E. Coumans, "Bullet physics simulation," in *ACM SIGGRAPH 2015 Courses*, 2015, p. 1.

[16] J. Liang, V. Makoviychuk, A. Handa, N. Chentanez, M. Macklin, and D. Fox, "Gpu-accelerated robotic simulation for distributed reinforcement learning," in *Conference on Robot Learning*. PMLR, 2018, pp. 270–282.

[17] Bullet Physics Engine, "Bullet Physics Engine," http://www.bulletphysics.org, n.d., [Online; accessed 23-April-2025].

[18] E. Todorov, T. Erez, and Y. Tassa, "Mujoco: A physics engine for model-based control," in *2012 IEEE/RSJ international conference on intelligent robots and systems*. IEEE, 2012, pp. 5026–5033.

[19] Open Dynamics Engine, "Open Dynamics Engine," http://ode.org, n.d., [Online; accessed 23-April-2025].

[20] DART Physics Engine, "DART physics engine," http://dartsim.github.io, n.d., [Online; accessed 23-April-2025].

[21] PhysX Physics Engine, "PhysX Physics Engine," http://www.geforce.com/hardware/technology/physx, n.d., [Online; accessed 23-April-2025].

[22] Rojtberg, Pavel and Rogers, David and Streeting, Steve and others, "Ogre scene-oriented, flexible 3d engine," https://www.ogre3d.org/, 2001 – 2024.

[23] S. G. Parker, J. Bigler, A. Dietrich, H. Friedrich, J. Hoberock, D. Luebke, D. McAllister, M. McGuire, K. Morley, A. Robison *et al.*, "Optix: a general purpose ray tracing engine," *Acm transactions on graphics (tog)*, vol. 29, no. 4, pp. 1–13, 2010.

[24] Unity Technologies, "Unity Game Engine," https://unity.com/, n.d., [Online; accessed 23-April-2025].

[25] Epic Games, "Unreal Engine," https://www.unrealengine.com/en-US, n.d., [Online; accessed 23-April-2025].

[26] *Simulink: Simulation and Model-Based Design*, https://www.mathworks.com/products/simulink.html, The MathWorks, Inc., Natick, Massachusetts, United States, 2023, [Accessed: 24-Feb-2025].

[27] R. Featherstone, *Rigid body dynamics algorithms*. Springer, 2014.

[28] N. Corporation, "Physx sdk," https://github.com/NVIDIAGameWorks/PhysX, 2025, accessed: 2025-04-26.

[29] Havok. (2025) Power to the creators. Accessed: 2025-04-26. [Online]. Available: https://www.havok.com/

[30] Y. Song, S. Naji, E. Kaufmann, A. Loquercio, and D. Scaramuzza, "Flightmare: A flexible quadrotor simulator," in *Conference on Robot Learning*. PMLR, 2021, pp. 1147–1157.

[31] F. Furrer, M. Burri, M. Achtelik, and R. Siegwart, "Rotors—a modular gazebo mav simulator framework," *Robot Operating System (ROS) The Complete Reference (Volume 1)*, pp. 595–625, 2016.

[32] J. Meyer, A. Sendobry, S. Kohlbrecher, U. Klingauf, and O. Von Stryk, "Comprehensive simulation of quadrotor uavs using ros and gazebo," in *Simulation, Modeling, and Programming for Autonomous Robots: Third International Conference, SIMPAR 2012, Tsukuba, Japan, November 5-8, 2012. Proceedings 3.* Springer, 2012, pp. 400–411.

[33] F. Mahmuddin, "Rotor blade performance analysis with blade element momentum theory," *Energy Procedia*, vol. 105, pp. 1123–1129, 2017.

[34] L. Bauersfeld, E. Kaufmann, P. Foehn, S. Sun, and D. Scaramuzza, "Neurobem: Hybrid aerodynamic quadrotor model," *arXiv preprint arXiv:2106.08015*, 2021.

[35] "Gazebo - staging.gazebosim.org," https://staging.gazebosim.org/, [Accessed 01-02-2024].

[36] M. A. Sherman, A. Seth, and S. L. Delp, "Simbody: multibody dynamics for biomedical research," *Procedia Iutam*, vol. 2, pp. 241–261, 2011.

[37] S. J. Pan and Q. Yang, "A survey on transfer learning," *IEEE Transactions on Knowledge and Data Engineering*, vol. 22, no. 10, pp. 1345–1359, 2010.

[38] H. Nam, H. Lee, J. Park, W. Yoon, and D. Yoo, "Reducing domain gap by reducing style bias," in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2021, pp. 8690–8699.

[39] S. Höfer, K. Bekris, A. Handa, J. C. Gamboa, M. Mozifian, F. Golemo, C. Atkeson, D. Fox, K. Goldberg, J. Leonard *et al.*, "Sim2real in robotics and automation: Applications and challenges," *IEEE transactions on automation science and engineering*, vol. 18, no. 2, pp. 398–400, 2021.

[40] J. Tobin, R. Fong, A. Ray, J. Schneider, W. Zaremba, and P. Abbeel, "Domain randomization for transferring deep neural networks from simulation to the real world," in *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2017, pp. 23–30.

[41] L. Bauersfeld, E. Kaufmann, P. Foehn, S. Sun, and D. Scaramuzza, "Neurobem: Hybrid aerodynamic quadrotor model," *arXiv preprint arXiv:2106.08015*, 2021.

[42] C. Martínez, I. F. Mondragón, M. A. Olivares-Méndez, and P. Campoy, "On-board and ground visual pose estimation techniques for uav control," *Journal of Intelligent & Robotic Systems*, vol. 61, pp. 301–320, 2011.

[43] V. Grabe, H. H. Bülthoff, and P. R. Giordano, "On-board velocity estimation and closed-loop control of a quadrotor uav based on optical flow," in *2012 IEEE International Conference on Robotics and Automation*. IEEE, 2012, pp. 491–497.

[44] V. Guizilini and F. Ramos, "Visual odometry learning for unmanned aerial vehicles," in *2011 IEEE International Conference on Robotics and Automation*. IEEE, 2011, pp. 6213–6220.

[45] A. Carrio, C. Sampedro, A. Rodriguez-Ramos, and P. Campoy, "A review of deep learning methods and applications for unmanned aerial vehicles," *Journal of Sensors*, vol. 2017, 2017.

[46] L. Tai and M. Liu, "Deep-learning in mobile robotics-from perception to control systems: A survey on why and why not," *arXiv preprint arXiv:1612.07139*, vol. 1, 2016.

[47] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 770–778.

[48] S. Ren, K. He, R. Girshick, and J. Sun, "Faster r-cnn: Towards real-time object detection with region proposal networks," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 39, no. 6, pp. 1137–1149, 2017.

[49] L.-C. Chen, G. Papandreou, I. Kokkinos, K. Murphy, and A. L. Yuille, "Deeplab: Semantic image segmentation with deep convolutional nets, atrous convolution, and fully connected crfs," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 40, no. 4, pp. 834–848, 2018.

[50] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *arXiv preprint arXiv:1707.06347*, 2017.

[51] J. Schulman, S. Levine, P. Abbeel, M. Jordan, and P. Moritz, "Trust region policy optimization," in *International conference on machine learning*. PMLR, 2015, pp. 1889–1897.

[52] D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, and M. Riedmiller, "Deterministic policy gradient algorithms," in *International conference on machine learning*. Pmlr, 2014, pp. 387–395.

[53] T. Haarnoja, A. Zhou, K. Hartikainen, G. Tucker, S. Ha, J. Tan, V. Kumar, H. Zhu, A. Gupta, P. Abbeel *et al.*, "Soft actor-critic algorithms and applications," *arXiv preprint arXiv:1812.05905*, 2018.

[54] J. Hwangbo, I. Sa, R. Siegwart, and M. Hutter, "Control of a quadrotor with reinforcement learning," *IEEE Robotics and Automation Letters*, vol. 2, no. 4, pp. 2096–2103, 2017.

[55] N. O. Lambert, D. S. Drew, J. Yaconelli, S. Levine, R. Calandra, and K. S. Pister, "Low-level control of a quadrotor with deep model-based reinforcement learning," *IEEE Robotics and Automation Letters*, vol. 4, no. 4, pp. 4224–4230, 2019.

[56] E. Kaufmann, L. Bauersfeld, A. Loquercio, M. Müller, V. Koltun, and D. Scaramuzza, "Champion-level drone racing using deep reinforcement learning," *Nature*, vol. 620, no. 7976, pp. 982–987, 2023.

[57] H. Wang, J. Xing, N. Messikommer, and D. Scaramuzza, "Environment as policy: Learning to race in unseen tracks," *arXiv preprint arXiv:2410.22308*, 2024.

[58] J. Panerati, H. Zheng, S. Zhou, J. Xu, A. Prorok, and A. P. Schoellig, "Learning to fly—a gym environment with pybullet physics for reinforcement learning of multi-agent quadcopter control," in *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2021, pp. 7512–7519.

[59] B. Xu, F. Gao, C. Yu, R. Zhang, Y. Wu, and Y. Wang, "Omnidrones: An efficient and flexible platform for reinforcement learning in drone control," *IEEE Robotics and Automation Letters*, vol. 9, no. 3, pp. 2838–2844, 2024.

[60] J. Eschmann, D. Albani, and G. Loianno, "Learning to fly in seconds," *IEEE Robotics and Automation Letters*, vol. 9, no. 7, pp. 6336–6343, 2024.

[61] N.-L. Nguyen and D. E. Meltzer, "Visualization tool for 3-d relationships and the right-hand rule," *The Physics Teacher*, vol. 43, no. 3, pp. 155–157, 2005.

[62] D. Scaramuzza and F. Fraundorfer, "Visual odometry [tutorial]," *IEEE robotics & automation magazine*, vol. 18, no. 4, pp. 80–92, 2011.

[63] E. Grood and W. Suntay, "A joint coordinate system for the clinical description of three-dimensional motions: Application to the knee," *Journal of Biomechanical Engineering*, vol. 105, no. 2, pp. 136–144, 1983.

[64] A. Mokhtari and A. Benallegue, "Dynamic feedback controller of euler angles and wind parameters estimation for a quadrotor unmanned aerial vehicle," in *IEEE International Conference on Robotics and Automation, 2004. Proceedings. ICRA'04. 2004*, vol. 3. IEEE, 2004, pp. 2359–2366.

[65] H. A. Hashim, "Special orthogonal group so (3), euler angles, angle-axis, rodriguez vector and unit-quaternion: Overview, mapping and challenges," *arXiv preprint arXiv:1909.06669*, 2019.

[66] M. W. Spong and M. Vidyasagar, *Robot dynamics and control*. John Wiley & Sons, 2008.

[67] C. Forster, L. Carlone, F. Dellaert, and D. Scaramuzza, "On-manifold preintegration for real-time visual–inertial odometry," *IEEE Transactions on Robotics*, vol. 33, no. 1, pp. 1–21, 2016.

[68] F. Lizarralde and J. T. Wen, "Attitude control without angular velocity measurement: A passivity approach," *IEEE transactions on Automatic Control*, vol. 41, no. 3, pp. 468–472, 1996.

[69] C. G. Mayhew, R. G. Sanfelice, and A. R. Teel, "Quaternion-based hybrid control for robust global attitude tracking," *IEEE Transactions on Automatic control*, vol. 56, no. 11, pp. 2555–2566, 2011.

[70] D. Choukroun, I. Y. Bar-Itzhack, and Y. Oshman, "Novel quaternion kalman filter," *IEEE Transactions on Aerospace and Electronic Systems*, vol. 42, no. 1, pp. 174–190, 2006.

[71] J. L. Marins, X. Yun, E. R. Bachmann, R. B. McGhee, and M. J. Zyda, "An extended kalman filter for quaternion-based orientation estimation using marg sensors," in *Proceedings 2001 IEEE/RSJ International Conference on Intelligent Robots and Systems. Expanding the Societal Role of Robotics in the the Next Millennium (Cat. No. 01CH37180)*, vol. 4. IEEE, 2001, pp. 2003–2011.

[72] Y.-J. Cheon and J.-H. Kim, "Unscented filtering in a unit quaternion space for spacecraft attitude estimation," in *2007 IEEE International Symposium on Industrial Electronics*. IEEE, 2007, pp. 66–71.

[73] M. T. Hussein and M. N. Nemah, "Modeling and control of quadrotor systems," in *2015 3rd RSI International Conference on Robotics and Mechatronics (ICROM)*. IEEE, 2015, pp. 725–730.

[74] T. A. Johansen and T. I. Fossen, "Control allocation—a survey," *Automatica*, vol. 49, no. 5, pp. 1087–1103, 2013.

[75] J. Eschmann, D. Albani, and G. Loianno, "Learning to fly in seconds," *IEEE Robotics and Automation Letters*, vol. 9, no. 7, pp. 6336–6343, 2024.

[76] H. Huang, G. M. Hoffmann, S. L. Waslander, and C. J. Tomlin, "Aerodynamics and control of autonomous quadrotor helicopters in aggressive maneuvering," in *2009 IEEE international conference on robotics and automation*. IEEE, 2009, pp. 3277–3282.

[77] P. Martin and E. Salaün, "The true role of accelerometer feedback in quadrotor control," in *2010 IEEE international conference on robotics and automation*. IEEE, 2010, pp. 1623–1629.

[78] R. Beard, "Quadrotor dynamics and control rev 0.1," 2008.

[79] Open Robotics, "Gazebo Sim API Reference: Class Index (version 9.1.0)," https://gazebosim.org/api/sim/9/classes.html, 2025, accessed: 2025-05-26.

[80] L. Hewing, J. Kabzan, and M. N. Zeilinger, "Cautious model predictive control using gaussian process regression," *IEEE Transactions on Control Systems Technology*, vol. 28, no. 6, pp. 2736–2743, 2019.

[81] R. A. Maller, G. Müller, and A. Szimayer, "Ornstein–uhlenbeck processes and extensions," *Handbook of financial time series*, pp. 421–437, 2009.

[82] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," *arXiv preprint arXiv:1509.02971*, 2015.

[83] L. C. Evans, *An introduction to stochastic differential equations*. American Mathematical Soc., 2012, vol. 82.

[84] R. P. Borase, D. Maghade, S. Sondkar, and S. Pawar, "A review of pid control, tuning methods and applications," *International Journal of Dynamics and Control*, vol. 9, pp. 818–827, 2021.

[85] T. Lee, M. Leok, and N. H. McClamroch, "Geometric tracking control of a quadrotor uav on se (3)," in *49th IEEE conference on decision and control (CDC)*. IEEE, 2010, pp. 5420–5425.

[86] R. Verschueren, G. Frison, D. Kouzoupis, J. Frey, N. v. Duijkeren, A. Zanelli, B. Novoselnik, T. Albin, R. Quirynen, and M. Diehl, "acados—a modular open-source framework for fast embedded optimal control," *Mathematical Programming Computation*, vol. 14, no. 1, pp. 147–183, 2022.

[87] J. A. Andersson, J. Gillis, G. Horn, J. B. Rawlings, and M. Diehl, "Casadi: a software framework for nonlinear optimization and optimal control," *Mathematical Programming Computation*, vol. 11, pp. 1–36, 2019.

[88] E. Gallo, "The so (3) and se (3) lie algebras of rigid body rotations and motions and their application to discrete integration, gradient descent optimization, and state estimation," *arXiv preprint arXiv:2205.12572*, 2022.

[89] J. Kang, J. Shan, and H. Alkomy, "Control framework for a uav slung-payload transportation system," *IEEE Control Systems Letters*, vol. 7, pp. 2473–2478, 2023.

[90] W. Guerra, E. Tal, V. Murali, G. Ryou, and S. Karaman, "Flightgoggles: Photorealistic sensor simulation for perception-driven robotics using photogrammetry and virtual reality," in *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2019, pp. 6941–6948.

[91] G. Silano, E. Aucone, and L. Iannelli, "Crazys: a software-in-the-loop platform for the crazyflie 2.0 nano-quadcopter," in *2018 26th Mediterranean Conference on Control and Automation (MED)*. IEEE, 2018, pp. 1–6.

[92] A. Garg, H.-T. L. Chiang, S. Sugaya, A. Faust, and L. Tapia, "Comparison of deep reinforcement learning policies to formal methods for moving obstacle avoidance," in *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2019, pp. 3534–3541.

[93] L. Tai, G. Paolo, and M. Liu, "Virtual-to-real deep reinforcement learning: Continuous control of mobile robots for mapless navigation," in *2017 IEEE/RSJ international conference on intelligent robots and systems (IROS)*. IEEE, 2017, pp. 31–36.

[94] O. Zhelo, J. Zhang, L. Tai, M. Liu, and W. Burgard, "Curiosity-driven exploration for mapless navigation with deep reinforcement learning," *arXiv preprint arXiv:1804.00456*, 2018.

[95] J. Hwangbo, J. Lee, A. Dosovitskiy, D. Bellicoso, V. Tsounis, V. Koltun, and M. Hutter, "Learning agile and dynamic motor skills for legged robots," *Science Robotics*, vol. 4, no. 26, p. eaau5872, 2019.

[96] V. Talpaert, I. Sobh, B. R. Kiran, P. Mannion, S. Yogamani, A. El-Sallab, and P. Perez, "Exploring applications of deep reinforcement learning for real-world autonomous driving systems," *arXiv preprint arXiv:1901.01536*, 2019.

[97] J. Li, L. Yao, X. Xu, B. Cheng, and J. Ren, "Deep reinforcement learning for pedestrian collision avoidance and human-machine cooperative driving," *Information Sciences*, vol. 532, pp. 110–124, 2020.

[98] X. Wang, S. Wang, X. Liang, D. Zhao, J. Huang, X. Xu, B. Dai, and Q. Miao, "Deep reinforcement learning: A survey," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 35, no. 4, pp. 5064–5078, 2024.

[99] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *Proceedings of the 44th annual international symposium on computer architecture*, 2017, pp. 1–12.

[100] M. L. Puterman, *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons, 2014.

[101] L. P. Kaelbling, M. L. Littman, and A. R. Cassandra, "Planning and acting in partially observable stochastic domains," *Artificial intelligence*, vol. 101, no. 1-2, pp. 99–134, 1998.

[102] A. G. Barto, "Reinforcement learning: An introduction. by richard's sutton," *SIAM Rev*, vol. 6, no. 2, p. 423, 2021.

[103] R. Bellman, "Dynamic programming," *science*, vol. 153, no. 3731, pp. 34–37, 1966.

[104] R. A. Howard, "Dynamic programming and markov processes." 1960.

[105] R. S. Sutton, "Learning to predict by the methods of temporal differences," *Machine learning*, vol. 3, pp. 9–44, 1988.

[106] R. S. Sutton, D. McAllester, S. Singh, and Y. Mansour, "Policy gradient methods for reinforcement learning with function approximation," *Advances in neural information processing systems*, vol. 12, 1999.

[107] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[108] H. Van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double Q-learning," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 30, no. 1, 2016.

[109] M. Hausknecht and P. Stone, "Deep recurrent Q-learning for partially observable mdps," in *2015 AAAI Fall Symposium Series*, 2015.

[110] S. S. Du, Y. Luo, R. Wang, and H. Zhang, "Provably efficient q-learning with function approximation via distribution shift error checking oracle," *Advances in Neural Information Processing Systems*, vol. 32, 2019.

[111] J. Progsch, "Threadpool - a simple c++11 thread pool implementation," https://github.com/progschj/ThreadPool, 2012, accessed: 2025-06-27.

[112] C. R. Harris, K. J. Millman, S. J. Van Der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith *et al.*, "Array programming with numpy," *Nature*, vol. 585, no. 7825, pp. 357–362, 2020.

[113] PyTorch Team, "Pytorch reinforcement learning documentation," https://docs.pytorch.org/rl/stable/index.html, 2024, accessed: 2025-06-27.

[114] NVIDIA Corporation, "Nvidia tensorrt," https://developer.nvidia.com/tensorrt, 2024, accessed: 2025-06-27.

[115] ONNX Community, "ONNX: Open neural network exchange," https://github.com/onnx/onnx, 2024, accessed: 2024-06-29.

[116] D. Hanover, A. Loquercio, L. Bauersfeld, A. Romero, R. Penicka, Y. Song, G. Cioffi, E. Kaufmann, and D. Scaramuzza, "Autonomous drone racing: A survey," *IEEE Transactions on Robotics*, vol. 40, pp. 3044–3067, 2024.

6 Appendix

A ROS

Robot Operating System (ROS) is an open-source middleware framework designed to facilitate the development of robotic applications. It provides a structured communications layer above the host operating systems of a heterogeneous compute cluster. ROS offers standardized message-passing between processes, also referred to as nodes, tools for simulation and visualization, and a rich ecosystem of packages for perception, control, and navigation. While not a real-time operating system, ROS enables modular and reusable software architectures, making it widely adopted in both research and industry for rapid prototyping and deployment of robotic systems.

A.1 ROS Nodes

As part of our toolbox, we developed a set of standard ROS Nodes that enable integration of open-source ROS packages directly with the simulation. Specifically, these nodes can be categorized into two subsets: `Publishers` and `Subscribers`. The Publishers leverage the ROS transport mechanism to broadcast the sensor data (Image, IMU, Pointcloud, etc.) and the UAV states using the standard ROS messages. The `Subscribers` listen to messages published by the `Publishers` and implement the desired algorithm logic. For our toolbox, we implement a `Subscriber` to listen for control or actuator commands, and

sends it to the `CustomServer` to perform allocation. Furthermore, we also implement a `rosbag` recording node that records the entire simulation experiment, which includes the UAV states, their derivatives, applied controls, and the sensor data. This feature enables offline analysis of the experiment data and facilitates sharing it online.

The ROS graph that represents the nodes, as well as the `Subscribers` and `Publishers`, is shown in Fig. 1. Here, the `controller_publisher` node represents the user controller, which can be implemented using our control library. This node publishes to the `control_cmd` topic that carries the control command message in a `Float32MultiArray` ROS message. This node also subscribes to the `/uav_state_and_state_dot` topic, which contains the UAV states and their derivatives in a `Float32MultiArray` ROS message. The `controller_subscriber` node subscribes to the control commands from `control_cmd` topic and sends them to the `CustomServer` to perform control allocation. Furthermore, this node also publishes a set of serialized data through `serialized_states`, `serialized_states_dots`, and `serialized_controls` using the `String` ROS message. This facilitates low overhead and efficient data recording. These topics are subscribed to by the `rosbag_data_recorder_subscriber` node, which automatically appends a timestamp (a `Header` message) to the received data and writes it to a `rosbag` buffer. Finally, the `sensor_data_publisher` node interfaces with the `CustomServer` and publishes the available sensor data to unique topics (here `imu0` and `imu1` in `IMU` ROS message, `/camera/image_raw` in `Image` ROS message, `/camera/camera_info` in `CameraInfo` ROS message, and `range_scan` in `Range` ROS message) which are also subscribed by the `rosbag_data_recorder_subscriber` for recording.

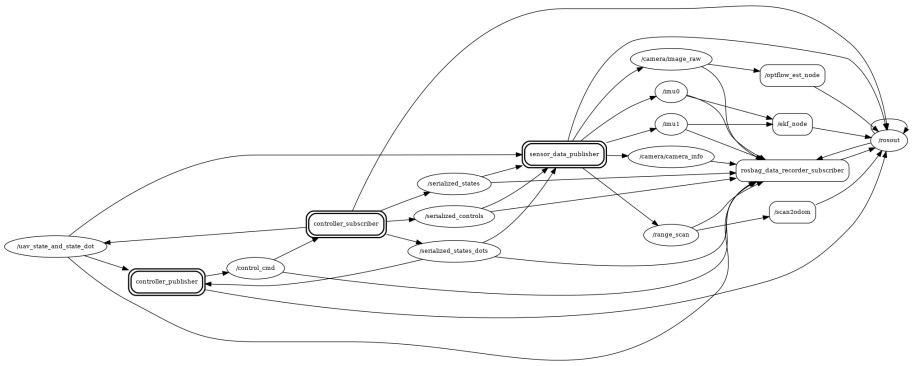

Figure 1: The ROS Nodes packaged with our toolbox

B SDF

The Simulation Description Format (SDF) is an XML-based specification used to define the structure, properties, and physical behavior of robots, objects, and environments for applications in simulation, visualization, and control. Developed for the Gazebo simulator to meet the requirements of research-oriented robotics, SDF has evolved into a robust and extensible standard. It supports the incorporation of kinematic and dynamic parameters, sensors, material properties, textures, and joint dynamics. SDF enables the construction of rich, interactive simulation environments, including varied terrain generated from fictional designs or real-world digital elevation models, and configurable lighting. External 3D assets can be integrated seamlessly, facilitating the rapid development of complex, realistic worlds in which robots can be tested, visualized, and controlled.

For our simulation toolbox, we implement an SDF model for UAVs as follows

```
<sdf version='1.11'>
  <model name='quadrotor'>
    <enable_wind>1</enable_wind>
    <link name='quadrotor/base_link'>
      <inertial>
        <pose>0 0 0 0 0 0</pose>
```

```
      <mass>1.5</mass>
      <inertia>
          <ixx>14387108.65e-9</ixx>
          <iyy>16559882.52e-9</iyy>
          <izz>28784890.45e-9</izz>
      </inertia>
  </inertial>
  <sensor name='rgb' type='camera'>
    ...
  </sensor>
  <sensor name='depth' type='camera'>
    ...
  </sensor>
  <sensor name='range' type='gpu_ray'>
    ...
  </sensor>
  <sensor name='imu0' type='imu'>
    ...
  </sensor>
  <sensor name='imu1' type='imu'>
    ...
  </sensor>
</link>

<joint name='quadrotor/rotor_0_joint' type='revolute'>
  <pose relative_to='quadrotor/base_link'>-0.1016 -0.127 0.023 0 0 0</pose>
  <parent>quadrotor/base_link</parent>
  <child>quadrotor/rotor_0</child>
  <axis>
    <xyz>0 0 1</xyz>
    <limit>
      <lower>-inf</lower>
      <upper>inf</upper>
    </limit>
    <dynamics>
      <spring_reference>0</spring_reference>
```

```
      <spring_stiffness>0</spring_stiffness>
    </dynamics>
  </axis>
</joint>
<link name='quadrotor/rotor_0'>
  <pose relative_to='quadrotor/rotor_0_joint'>0 0 0 0 0 0</pose>
  <inertial>
    <pose>0 0 0 0 0 0</pose>
    <mass>0.005</mass>
    <inertia>
      <ixx>1.9500000000000001e-07</ixx>
      <ixy>0</ixy>
      <ixz>0</ixz>
      <iyy>3.3340833333333338e-05</iyy>
      <iyz>0</iyz>
      <izz>3.3520833333333339e-05</izz>
    </inertia>
  </inertial>
</link>
<!--repeat for all rotors-->
...

<plugin name='control_library::RotorPlugin' filename='rotor_plugin'>
  <jointName>quadrotor/rotor_0_joint</jointName>
  <linkName>quadrotor/rotor_0</linkName>
  <turningDirection>ccw</turningDirection>
  <timeConstantUp>0.0005</timeConstantUp>
  <timeConstantDown>0.0005</timeConstantDown>
  <maxRotVelocity>3400.0</maxRotVelocity>
  <motorQuadraticParams>1.9533e-6 0.0008594 -0.0121</motorQuadraticParams>
  <airDragCoeffs>0.3 0.3 0.25</airDragCoeffs>
  <groundEffectConstant>0.0006</groundEffectConstant>
  <momentConstant>0.01454</momentConstant>
  <actuator_number>0</actuator_number>
  <rotorDragCoefficient>0.00020673</rotorDragCoefficient>
  <rollingMomentCoefficient>0</rollingMomentCoefficient>
```

```
    </plugin>
    <!--repeat for all rotors-->
    ...
  </model>
</sdf>
```

C RotorPlugin

`RotorPlugin` is a Gazebo system plugin that implements the rotor level allocation. This plugin is implemented using the Gazebo C++ API. Specifically, we inherit from the `ISystemConfigure` and `ISystemPreUpdate` interfaces to implement this plugin. At each simulation step, `CustomServer` sends the desired actuator command to `RotorPlugin`, and the goal of each `RotorPlugin` is to calculate the rotor-dependent nominal and disturbance dynamics and apply the forces at the location of the rotor and the moments at the center of gravity of the UAV.

C.1 `RotorPlugin ISystemConfigure`

In the `Configure` phase of the `RotorPlugin`, the Gazebo server provides the SDF element corresponding to the plugin (see Section B), and a reference to the Entity Component Manager (ECM) through which we can apply the wrench and obtain states of simulation entities. In this phase, the `RotorPlugin` stores references to the associated rotor link of the UAV. Finally, the first-order low-pass filter is instantiated to simulate the rate limit on actuator commands. Note that this phase is only executed once in the simulation when the plugin is first attached to the UAV.

The `Configure` method implementation of `RotorPlugin` is shown below:

```
void RotorPlugin::Configure(const Entity &_entity,
const std::shared_ptr<const sdf::Element> &_sdf,
EntityComponentManager &_ecm,
```

```
    EventManager &)
{
  this->model = Model(_entity);
  this->jointName = _sdf->Get<std::string>("jointName");
  this->linkName = _sdf->Get<std::string>("linkName");
  this->actuatorNumber = _sdf->GetElement("actuator_number")->Get<int>();
  auto turningDirection =
  _sdf->GetElement("turningDirection")->Get<std::string>();
  if (turningDirection == "cw")
    this->turningDirection = turning_direction::kCw;
  else if (turningDirection == "ccw")
    this->turningDirection = turning_direction::kCcw;
  _sdf->Get<double>("rotorDragCoefficient",
      this->rotorDragCoefficient,
      this->rotorDragCoefficient);
  _sdf->Get<double>("rollingMomentCoefficient",
      this->rollingMomentCoefficient,
      this->rollingMomentCoefficient);
  _sdf->Get<double>("maxRotVelocity",
      this->maxRotVelocity, this->maxRotVelocity);
  _sdf->Get<double>("momentConstant",
      this->momentConstant, this->momentConstant);
  _sdf->Get<double>("timeConstantUp",
      this->timeConstantUp, this->timeConstantUp);
  _sdf->Get<double>("timeConstantDown",
      this->timeConstantDown, this->timeConstantDown);
  _sdf->Get<double>("rotorVelocitySlowdownSim",
      this->rotorVelocitySlowdownSim, 10);
  _sdf->Get<math::Vector3d>("motorQuadraticParams",
      this->motorQuadraticParams, this->motorQuadraticParams);
  _sdf->Get<math::Vector3d>("airDragCoeffs",
      this->airDragCoeffs, this->airDragCoeffs);
  _sdf->Get<double>("groundEffectConstant",
      this->groundEffectConstant, this->groundEffectConstant);
  // Create the first order filter.
  this->rotorVelocityFilter = std::make_unique<FirstOrderFilter<double>>(
```

```
        this->timeConstantUp, this->timeConstantDown,
        this->refMotorInput);
}
```

C.2 RotorPlugin ISystemPreUpdate

The `PreUpdate` phase occurs just before each simulation step, and plugins that inherit the `ISystemPreUpdate` will have their implemented `PreUpdate` method invoked. This phase can be used to apply a wrench to simulation entities. The applied wrench will be utilized during the `Update` phase of the simulation to update the states of entities. In this phase, the server invokes the implementation of `PreUpdate` with a reference to the Entity Component Manager (ECM) that can be used to mutate components of entities in the simulation. Specifically, our implementation obtains the current UAV state, rotor state, and the wind velocity using the ECM, as well as the allocated rotor velocity from the `ServerInterface` to calculate the nominal and disturbance translation and attitude dynamics for each rotor. They are then applied to the UAV using the ECM.

The `PreUpdate` method implementation of our `RotorPlugin` is shown below.

```
void RotorPlugin::PreUpdate(const UpdateInfo &_info,
    EntityComponentManager &_ecm)
{
      double samplingTime = _info.dt;
      auto jointEntity = this->model.JointByName(_ecm, this->jointName);
      auto linkEntity = this->model.LinkByName(_ecm, this->linkName);
      auto parentLinkEntity = this->model.LinkByName(_ecm,
      ↪  _ecm.Component<components::ParentLinkName>(
        jointEntity););
      const auto jointVelocity = _ecm.Component<components::JointVelocity>(jointEntity);
      double motorRotVel = jointVelocity->Data()[0];
      int realMotorVelocitySign =
          (motorRotVel > 0) - (motorRotVel < 0);
```

```
double absMotorVel = realMotorVelocitySign*motorRotVel;
double thrust = absMotorVel*absMotorVel*this->motorQuadraticParams.X() +
                absMotorVel*this->motorQuadraticParams.Y()  +
                this->motorQuadraticParams.Z();
using Pose = math::Pose3d;
using Vector3 = math::Vector3d;
Link link(linkEntity);
const auto worldPose = link.WorldPose(_ecm);
auto heightFromGround = worldPose->Pos().Z();
double groundEffectForce =
↪  this->groundEffectConstant*thrust/(heightFromGround*heightFromGround);
link.AddWorldForce(_ecm, worldPose->Rot().RotateVector(Vector3(0, 0,
↪  thrust+groundEffectForce)));
const auto jointPose = _ecm.Component<components::Pose>(jointEntity);
Pose jointWorldPose = *worldPose * jointPose->Data();
const auto jointAxisComp = _ecm.Component<components::JointAxis>(jointEntity);
const auto worldLinearVel = link.WorldLinearVelocity(_ecm);
Entity windEntity = _ecm.EntityByComponents(components::Wind());
auto windLinearVel = _ecm.Component<components::WorldLinearVelocity>(windEntity);
Vector3 windSpeedWorld = windLinearVel->Data();

Vector3 jointAxis = jointWorldPose.Rot().RotateVector(jointAxisComp->Data().Xyz());
Vector3 bodyVelocityWorld = *worldLinearVel;
Vector3 relativeWindVelocityWorld = bodyVelocityWorld - windSpeedWorld;
Vector3 bodyVelocityPerpendicular = relativeWindVelocityWorld -
                                    (relativeWindVelocityWorld.Dot(jointAxis) * jointAxis);
Vector3 airDrag = -std::abs(motorRotVel) *
                        this->rotorDragCoefficient *
                        bodyVelocityPerpendicular;
link.AddWorldForce(_ecm, airDrag);
Vector3 parentWorldTorque;
auto parentWrenchComp = _ecm.Component<components::ExternalWorldWrenchCmd>( parentLinkEntity);
Link parentLink(parentLinkEntity);
const auto parentWorldPose = parentLink.WorldPose(_ecm);

const auto angularVelocityWorld = _ecm.Component<components::WorldAngularVelocity>(
```

```
      parentLinkEntity);
  auto angularVelocityBody =
  ↪  parentWorldPose->Rot().RotateVectorReverse(angularVelocityWorld->Data());
  const Vector3 negativeG = Vector3(-angularVelocityBody.Y(), angularVelocityBody.X(),0.0);
  const Vector3 gyroMoments =
  ↪  parentWorldPose->Rot().RotateVector(4.92e-6*negativeG*motorRotVel);

  Vector3 baseLinkLinearVel=
  ↪  _ecm.Component<components::WorldLinearVelocity>(parentLinkEntity)->Data();
  Vector3 airDragInertial = -baseLinkLinearVel*this->airDragCoeffs;
  parentLink.AddWorldForce(_ecm,airDragInertial);
  Pose poseDifference = (*parentWorldPose).Inverse() * (*worldPose);
  Vector3 dragTorque(
      0, 0, -this->turningDirection * thrust * this->momentConstant);
  Vector3 dragTorqueParentFrame =
      poseDifference.Rot().RotateVector(dragTorque);
  parentWorldTorque =
      parentWorldPose->Rot().RotateVector(dragTorqueParentFrame);

  Vector3 rollingMoment;
  rollingMoment = -std::abs(motorRotVel) *
                  this->rollingMomentCoefficient *
                  bodyVelocityPerpendicular;
  parentWorldTorque += rollingMoment;
  parentWorldTorque += gyroMoments;
  msgs::Set(parentWrenchComp->Data().mutable_torque(),
  msgs::Convert(parentWrenchComp->Data().torque()) + parentWorldTorque);
  double refMotorRotVel;
  refMotorRotVel = this->rotorVelocityFilter->UpdateFilter(
      this->refMotorInput, samplingTime);

      _ecm.SetComponentData<components::JointVelocityCmd>(
    jointEntity,
    {this->turningDirection * refMotorRotVel});
}
```

D ServerInterface

The server interface is the Gazebo system plugin that allocates the rotor velocity to each `RotorPlugin`. Furthermore, it also implements the functionalities of the Control library. It runs the controller at a desired control frequency and allocates the control inputs to `RotorPlugins` and obtains the necessary feedback information to pass to the controller. The `ServerInterface` implements `ISystemConfigure`, `ISystemPreUpdate`, and `ISystemPostUpdate` interfaces.

D.1 ServerInterface ISystemConfigure

In the `Configure` method, the `ServerInterface` obtains a reference to the UAV model entity. The implementation of `Configure` method of `ServerInterface` is shown below:

```
    void ServerInterface::Configure(
                const Entity &_entity,
                const std::shared_ptr<const sdf::Element> &_sdf,
                EntityComponentManager &_ecm,
                EventManager &_eventMgr)
{
  this->sdfCreator.reset();
  this->sdfCreator = std::make_unique<gz::sim::SdfEntityCreator>(_ecm, _eventMgr);
  this->mainParentEntity = _entity;
  auto entity = _ecm.EntityByName(this->linkName);
  this->link = gz::sim::Link(*entity);
  this->linkEntity = *entity;
  entity = _ecm.EntityByName(this->modelName);
  this->model = gz::sim::Model(*entity);
  this->modelEntity = *entity;
        this->link.EnableVelocityChecks(_ecm, true);
  this->link.EnableAccelerationChecks(_ecm, true);
}
```

D.2 ServerInterface ISystemPreUpdate

In `PreUpdate` phase, the `ServerInterface` accesses the current reference and invokes the controller to obtain actuator commands and sends them to the `RotorPlugins` in a serialized format. Note that if the custom controller doesn't implement the `calculate_actuator_cmd` method, then `CustomServer` automatically implements this method by first determining the allocation matrix based on the SDF description and wrapping the allocation algorithm over the `calculate_u` method. The `PreUpdate` implementation of `ServerInterface` is shown below. Note that the reference needs to be set by the user, and can be done by accessing the `reference` variable of `ServerInterface` through `CustomServer`. Similarly, the controller can also be set by accessing the `controller` variable of `ServerInterface` through `CustomServer`.

```
    void ServerInterface::PreUpdate(const UpdateInfo &_info,
      EntityComponentManager &_ecm)
{
      auto ref = this->reference;
      auto actuatorCmd = this->controller.calculate_actuator_cmd(this->state, this->stateDot, ref);
      for (int i = 0; i<this->actuator_dim; ++i){
        this->actuatorCmdSerialized.set_velocity(i, actuatorCmd.at(i));
      }
      auto actuatorMsgComp = _ecm.Component<components::Actuators>(this->modelEntity);
      auto compFunc = [](const msgs::Actuators &_a, const msgs::Actuators &_b){
              return std::equal(_a.velocity().begin(), _a.velocity().end(),
                                _b.velocity().begin());
           };
      auto state = actuatorMsgComp->SetData(this->actuatorCmdSerialized, compFunc)
                     ? ComponentState::PeriodicChange
                     : ComponentState::NoChange;
      _ecm.SetChanged(this->modelEntity, components::Actuators::typeId, state);
}
```

D.3 ServerInterface ISystemPostUpdate

In the `PostUpdate` phase, the `ServerInterface` obtains the feedback information using the ECM. Finally, it also obtains the sensor data and stores it in a serialized format in a `std::map` structure, which can be accessed using the `CustomServer`. The `PostUpdate` implementation of `ServerInterface` is shown below.

```
    void ServerInterface::PostUpdate(const UpdateInfo &_info,
    const EntityComponentManager &_ecm)
{
  auto poseOpt = this->link.WorldPose(_ecm);
  this->pose =  *poseOpt;
  auto velOpt = this->link.WorldLinearVelocity(_ecm);
  this->vel =  *velOpt;
  auto omegaOpt = this->link.WorldAngularVelocity(_ecm);
  this->omega = *omegaOpt;
  auto alphaOpt = this->link.WorldAngularAcceleration(_ecm);
  this->alpha =*alphaOpt;
  auto accelOpt = this->link.WorldLinearAcceleration(_ecm);
  this->accel = *accelOpt;
  this->poseData.at(0) = this->pose.Pos().X();
  this->poseData.at(1) = this->pose.Pos().Y();
  this->poseData.at(2) = this->pose.Pos().Z();
  this->poseData.at(3) = this->pose.Rot().W();
  this->poseData.at(4) = this->pose.Rot().X();
  this->poseData.at(5) = this->pose.Rot().Y();
  this->poseData.at(6) = this->pose.Rot().Z();
  this->poseData.at(7) = this->vel.X();
  this->poseData.at(8) = this->vel.Y();
  this->poseData.at(9) = this->vel.Z();
  this->poseData.at(10) = this->omega.X();
  this->poseData.at(11) = this->omega.Y();
  this->poseData.at(12) = this->omega.Z();
  this->poseData.at(13) = this->accel.X();
  this->poseData.at(14) = this->accel.Y();
```

```
  this->poseData.at(15) = this->accel.Z();
  this->poseData.at(16) = this->alpha.X();
  this->poseData.at(17) = this->alpha.Y();
  this->poseData.at(18) = this->alpha.Z();

  this->state[0] = this->poseData[0];this->state[1] = this->poseData[1];this->state[2] =
  ↪  this->poseData[2];
  this->state[3] = this->poseData[7];this->state[4] = this->poseData[8];this->state[5] =
  ↪  this->poseData[9];
  this->state[6] = this->poseData[3];this->state[7] = this->poseData[4];this->state[8] =
  ↪  this->poseData[5]; this->state[9] = this->poseData[6];
  this->state[10] = this->poseData[10];this->state[11] = this->poseData[11]; this->state[12] =
  ↪  this->poseData[12];

  this->stateDot[0] = this->poseData[7];this->stateDot[1] = this->poseData[8];this->stateDot[2] =
  ↪  this->poseData[9];
  this->stateDot[3] = this->poseData[13];this->stateDot[4] = this->poseData[14];this->stateDot[5] =
  ↪  this->poseData[15];
  Eigen::Quaterniond quat{this->state[6], this->state[7], this->state[8], this->state[9]};
  Eigen::Vector3d omega{this->state[10], this->state[11], this->state[12]};
  auto quatDeriv = quadrotor_controllers::quaternion_derivative(quat, omega);
  this->stateDot[6] = quatDeriv(0);this->stateDot[7] = quatDeriv(1);this->stateDot[8] =
  ↪  quatDeriv(2); this->stateDot[9] = quatDeriv(3);
  this->stateDot[10] = this->poseData[16];this->stateDot[11] = this->poseData[17];
  ↪  this->stateDot[12] = this->poseData[18];
  this->sensorDataMap = getSensorData(_ecm);

}
```

E CustomServer

E.1 Gazebo Server

In the default Gazebo setup, a `Server` and a `GUI` run in parallel in two processes (Fig. 2).

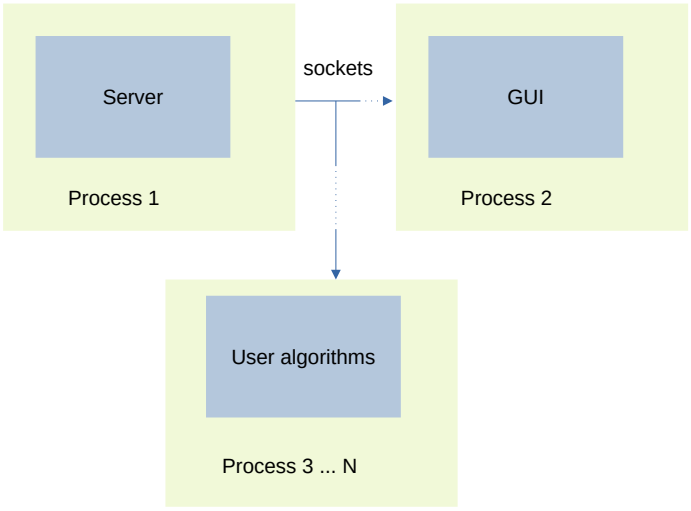

Figure 2: The Server-GUI-User processes

To overcome the limitations of this architecture, we redesign this to run Gazebo without any IPC dependence when interfacing with the user side. This is achieved by running a Server in a specialized C++ class that attaches a `ServerInterface` plugin to the Server. This new design leverages the `ServerInterface` to interact with the underlying server (Fig. 3).

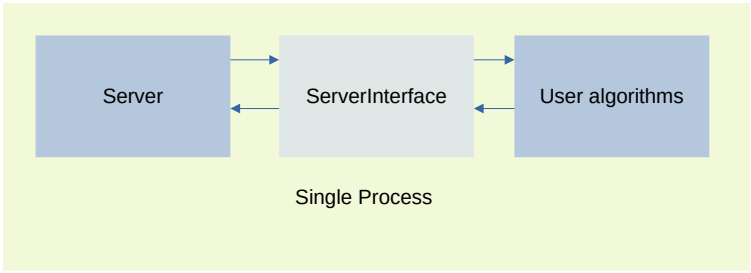

Figure 3: The `CustomServer` mediated architecture

E.2 Implementation of `CustomServer`

The CustomServer is implemented leveraging the Gazebo simulator C++ API. This server attaches the `ServerInterface` plugin into the simulation world as well as orchestrates the control allocation algorithm. The `RotorPlugin` system plugin is not directly created by the `CustomServer`, and it is specified through an SDF file that describes the UAV rigid body. This is intentionally performed so that the user can modify the UAV type as desired and attach the necessary number of rotor plugins, without needing to modify the behavior in `CustomServer`. The `CustomServer` is also responsible for automatically implementing the allocation algorithm.

The implementation of `CustomServer` is as follows :

```
class CustomServer
{
  public:
    CustomServer(const std::string& partition, const std::string &sdf_file, const std::string
    ↪  &model_name,
        const std::string &link_name, const std::string& world_name);
    ~CustomServer();
    void runOnce();
    void runN(int N);
```

```
    std::array<double, 19> state_info();
    void setController(Controller controller);
    void setReference(Stated reference);
    void setCollisionToTrack(std::string name);
    std::shared_ptr<Controller> controller;
    std::tuple<std::array<double, 13>, std::array<double, 13>> controlStates;
    std::string modelName;
    std::string linkName;
    std::unique_ptr<gz::sim::Server> server{nullptr};
    std::shared_ptr<ServerInterface> internalSys{nullptr};
    std::unique_ptr<gz::sim::ServerConfig> serverConfig{nullptr};
    std::string identity;
    const std::string _partition;
    const std::string _sdf_file;
    const std::string _model_name;
    const std::string _link_name;
    const std::string _world_name;
};


    CustomServer::CustomServer(const std::string& partition, const std::string &sdf_file, const
    ↪  std::string &model_name,
  const std::string &link_name, const std::string& world_name):
  _partition(partition),_sdf_file(sdf_file),_model_name(model_name),
  _link_name(link_name), _world_name(world_name){


  this->server.reset();
  this->serverConfig.reset();
  this->internalSys.reset();
  gz::common::Console::SetVerbosity(1);
  serverConfig = std::make_unique<gz::sim::ServerConfig>();
  serverConfig->SetSdfFile(this->_sdf_file);
  double updateRate = 20000;
  serverConfig->SetUpdateRate(updateRate);
  serverConfig->SetHeadlessRendering(true);
  serverConfig->SetUseLevels(true);
```

```
this->modelName = this->_model_name;
this->linkName = this->_link_name;
this->server = std::make_unique<gz::sim::Server>(*serverConfig);
this->internalSys = std::make_shared<ServerInterface>(this);
this->server->AddSystem(this->internalSys);
this->controller = controllers::getControllerImpl();
this->controlStates = {std::array<double, 13>(), std::array<double, 13>()};
}

void CustomServer::runOnce(){
  this->server->Run(true , 1 , false);
}
void CustomServer::runN(int N){
  this->server->Run(true , N , false);
}

void setController(Controller controller){
  try
  {
    Stated test;
    controller.calculate_actuator_cmd(test, test, test);
    this->internalSys->controller = controller;
  }
  catch (const std::runtime_error& e) {
  auto allocationMatrix = allocationMatrixComputer(this->rotorPositions, this->forceConstant,
  ↪  this->momentConstant);
  auto invAllocationMatrix = allocationMatrix.completeOrthogonalDecomposition().pseudoInverse();
  class NewController: public Controller{
      public:
        NewController(){
          this->_controller = controller;
        };
        ~NewController();
        Eigen::MatrixXd calculate_u(Stated& state, Stated& state_dot, Stated& desired_state)
        ↪  override
        {
```

```
            return this->_controller.calculate_u(state, state_dot, desired_state);
          };
          Eigen::MatrixXd calculate_actuator_cmd(Stated& state, Stated& state_dot, Stated&
          ↪  desired_state) override{
            auto u = this->_controller.calculate_u(state, state_dot, desired_state);
            return invAllocationMatrix*u;
          };
          Controller _controller;
    };
      this->internalSys->controller = NewController();
    }


  }

void CustomServer::setReference(Stated ref){
  this->internalSys->reference = ref;
}
```

The automatic allocation matrix computing is implemented as follows:

```
 Eigen::Matrix4Xd allocationMatrixComputer(Eigen::MatrixXd& rotorPositions ,
  double forceConstant, double momentConstant){
  Eigen::Matrix4Xd allocationMatrix;
  allocationMatrix.resize(4, rotorPositions.rows());
  unsigned int i = 0;
  for (int i=0; i< rotorPositions.rows(); ++i)
  {
    auto rotorPos = rotorPositions.row(i);
    int direction = rotorPos(2);
    double rollMomentArm = rotorPos(1);
    double pitchMomentArm = rotorPos(0);
    allocationMatrix(0, i) = 1;
    allocationMatrix(1, i) = rollMomentArm;
```

```
    allocationMatrix(2, i) =-pitchMomentArm;
    allocationMatrix(3, i) = -direction * momentConstant;

  }

  return forceConstant*allocationMatrix;
}
```

F DRLServer

`DRLServer` leverages the capabilities of `CustomServer` to interact with simulation entities. Furthermore, it also implements a set of APIs that make it suitable for Deep Reinforcement Learning applications. These APIs follow the Envpool and OpenAI Gym specifications. The implementation of `DRLServer` and the base environment is as follows. Note that custom environments can be created by inheriting the base environment and implementing the interface methods:

```
class DRLServerSpecs {
 public:
  template <typename Config>
  static decltype(auto) StateSpec(const Config& conf) {
    float fmax = std::numeric_limits<float>::max();
    int observation_dim = OBSERVATION_DIM;
    std::vector<float> obs_min(observation_dim, -fmax);
    std::vector<float> obs_max(observation_dim, 1.0f);
    for (int i = 0; i < (int)obs_min.size(); i++) {
      obs_max[i] = -obs_min[i];
    }
    return MakeDict("obs"_.Bind(Spec<float>({observation_dim}, {obs_min, obs_max})));
  }
  template <typename Config>
```

```
  static decltype(auto) ActionSpec(const Config& conf) {
    return MakeDict("action"_.Bind(Spec<float>({-1, ACTION_DIM}, {-1.0f, 1.0f})));
  }
};

using DRLServerEnvSpec = EnvSpec<DRLServerSpecs>;
class DRLServerEnv : public Env<DRLServerEnvSpec> {
 public:
  std::unique_ptr<DRLServer> server;
  const Stated defaultState{0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0};
  bool done{true};
  double physicsTimeStep = PHYSICS_TIME_STEP;
  double controlTimestep = CONTROL_TIME_STEP;
  int NStepsPerControl;
  int maxSteps = MAX_STEPS;
  int currentStep;
  std::shared_ptr<Controller> controller;
  float last_reward = 0.0;
  std::string partition;

  DRLServerEnv(const Spec& spec, int envid)
      : Env<DRLServerEnvSpec>(spec, envid) {
      this->partition = std::to_string(envid);
      const std::string sdf_file = ("world.sdf").string();
      const std::string model_name = "quadrotor";
      const std::string link_name = "quadrotor/base_link";
      const std::string world_name = "world_demo";
      this->server = std::make_unique<DRLServer>(this-<partition, sdf_file, model_name, link_name,
      ↪  world_name);
      this->NStepsPerControl = static_cast<int>(controlTimestep / physicsTimeStep);
      this->currentStep = 0;
      this->server->runOnce();
      this->server->setController(this->controller);
    }
  void WriteState(float reward) {
```

```
  State state = Allocate();
  ...
}
bool IsDone() override { return this->done; }

void Reset() override {

  this->partition = std::to_string(envid);
  const std::string sdf_file = ("world.sdf").string();
  const std::string model_name = "quadrotor";
  const std::string link_name = "quadrotor/base_link";
  const std::string world_name = "world_demo";
  this->server = std::make_unique<DRLServer>(this-<partition, sdf_file, model_name, link_name,
  ↪  world_name);
  this->NStepsPerControl = static_cast<int>(controlTimestep / physicsTimeStep);
  this->currentStep = 0;
  this->server->runOnce();
  this->server->setController(this->controller);
}

void Step(const Action& action) override {

  this->done = (++this->currentStep >= this->maxSteps);
  float* action_data = static_cast<float*>(action["action"_].Data());
  this->stateInfo = std::get<0>(this->server->controlStates);
  Stated desiredState;
  desiredState.setZero();
  desiredState.segment<3>(0) = this->stateInfo.segment<3>(0).cast<double>() +
                               action_array.segment<3>(0).cast<double>();
  float desired_yaw = 0.0;
  desiredState(6) = std::cos(desired_yaw / 2.0);
  desiredState(9) = std::sin(desired_yaw / 2.0);

  this->server->setReference(desiredState);
  this->server->runN(this->nStepsPerControl);
  this->stateInfo = std::get<0>(this->server->controlStates);
```

```
    float reward = this->getReward();
    this->WriteState(reward);
  }

  float getReward() {
    ...
  };

using DRLServerEnvpool = AsyncEnvPool<DRLServerEnv>;
int DRLServerEnv::env_id = 0;
std::mutex DRLServerEnv::lock = std::mutex();
std::vector<int> DRLServerEnv::env_locks = std::vector<int>();

}
```

List of Publications

1. Y. Liu, **A. Haridevan**, H. Schofield and J. Shan, "Application of Ghost-DeblurGAN to Fiducial Marker Detection," IEEE/RSJ International Conference on Intelligent Robots and Systems (**IROS**), Kyoto, Japan, 2022, pp. 6827-6832, DOI: 10.1109/IROS47612.2022.9981701.

2. **A. Haridevan**, J. Kang, M. Yuan, and J. Shan, "ROS2-Gazebo Simulator for Drone Applications," in *Proc. 2024 Int. Conf. Unmanned Aircraft Systems (ICUAS)*, pp. 1232–1238, IEEE, 2024.

3. Y. Liu, J. Shan, **A. Haridevan**, and S. Zhang, "L-PR: Exploiting LiDAR Fiducial Marker for Unordered Low Overlap Multiview Point Cloud Registration," IEEE Transactions on Instrumentation & Measurement (**TIM**), in press, 2025.

4. T. Liao, **A. Haridevan**, Y. Liu, and J. Shan, "Autonomous Vision-Based UAV Landing with Collision Avoidance Using Deep Learning," Science and Information Conference (**SAI**), pp. 79–87, 2022, DOI: 10.1007/978-3-031-10464-0_6.