Simulation & Integration of a 6-DOF Controllable Multirotor Vehicle

Collin A. Deans

Thesis submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

Master of Science

in

Aerospace Engineering

Jonathan T. Black, Chair

Daniel D. Doyle

Mathieu Joerger

June 18th, 2020

Blacksburg, Virginia

Keywords: Multi-rotor vehicles, Spacecraft simulation, 6-DOF, Dynamics simulator

Simulation & Integration of a 6-DOF Controllable Multirotor Vehicle

Collin A. Deans

(ABSTRACT)

The purpose of this thesis is to develop an existing design of a fully controllable multi-rotor vehicle toward simulating small satellite dynamics, enabling technology development to be accelerated and component failure risks to be mitigated by providing a testing platform with dynamics similar to those of small satellites in orbit. Evaluating dynamics-sensitive software and hardware components for use in small satellite operations has typically been relegated to simulated or physically constrained testing environments. More recently, researchers have begun using multi-rotor aerial vehicles to mimic the orbital motion of such satellites, further increasing simulation fidelity. The dynamical nature of multi-rotor vehicles allows them to accurately simulate the translational dynamics of a small satellite, but they struggle to accurately simulate rotational dynamics, as conventional multi-rotor vehicles' translational and rotational dynamics are coupled. In this thesis, an optimal design for a multi-rotor vehicle independently controllable in all six degrees of freedom is evaluated as a suitable simulation platform. The design of the proposed physical system is discussed and progress toward its construction is demonstrated. To facilitate future research endeavors, a simulation of the vehicle in a software-in-the-loop environment, using the Gazebo dynamics simulator, is developed and its performance evaluated. This simulation is then used to evaluate the vehicle's feasibility as a small-satellite dynamics simulator by tasking it with tracking dynamic position and attitude time histories representative of a small satellite.

Simulation & Integration of a 6-DOF Controllable Multirotor Vehicle

Collin A. Deans

(GENERAL AUDIENCE ABSTRACT)

When developing a spacecraft, it can be difficult to accurately test software and hardware that are sensitive to the spacecraft's motion. This difficulty arises because the space environment experienced by orbiting spacecraft allows them to move and rotate freely, and recreating this freedom of motion on earth requires large, expensive, and difficult-to-access test equipment. To make this testing more accessible, researchers have begun using quadcopter drones to mimic some aspects of a spacecraft's motion. While quadcopters can move like an orbiting spacecraft can, their designs do not allow them to rotate like an orbiting spacecraft can, thus providing an incomplete recreation of spacecraft motion. To correct this shortcoming, an existing drone design that is able to move and rotate simultaneously without fear of crashing is developed, with progress shown toward its construction. A software simulation of the drone is developed to help future researchers test software and algorithms before flying it on the physical drone. The simulation is then used to see how well the drone design can recreate the motions that a small spacecraft would experience.

Dedication

To my Saviour, for His unending grace.

To my wife, for her love, patience and encouragement.

To all of my friends, for supporting me and reminding me not to work too hard.

Acknowledgments

I would like to thank my committee, Dr. Black, Dr. Doyle, and Dr. Joerger, for their support and guidance during the development of this thesis. Specifically, I would like to thank Dr. Doyle for meeting with me weekly and providing one-on-one mentorship during the duration of my time as a graduate student with the Hume center. His guidance and advice were invaluable in the completion of this work.

I would also like to thank my fellow graduate students and the researchers at the Hume center and with Space@VT, specifically Gustavo Gargioni, Jeremy Ogorzalek, Theresa Blandino, and the rest of the SpaceDrones group, without whose knowledge and support I would still be stuck on one of the numerous obstacles encountered during the development of this work.

Contents

List of Figures

List of Tables

List of Abbreviations

ENU East-north-up

FCU Flight Control Unit

ISS International Space Station

NED North-east-down

ROS Robotics Operating System

SITL Software-in-the-loop

Chapter 1

Introduction

1.1 Motivation

Recently, small satellites, or "smallsats", have become increasingly accessible as a means of performing research and testing hardware and software components in the space environment. Access to spaceborne hardware has previously been held by private industry and governmental agencies only, as the expense of sending hardware to space priced out many smaller institutions. Now, due to the ease of constructing standardized platforms such as cubesats, and thanks to the prevalence of educational outreach and ride-share programs within the space launch industry, there are many opportunities for small companies, universities, and even grade schools to construct and send their own smallsats and payloads into orbit.

One such program is NASA's CubeSat Launch Initiative, or CSLI, which "provides access to space for small satellites, CubeSats, developed by the NASA Centers and programs, educational institutions and non-profit organizations giving CubeSat developers access to a

low-cost pathway to conduct research in the areas of science, exploration, technology development, education or operations" [1]. The CSLI functions by providing rideshare opportunities on select Educational Launch of Nanosatellite (ELaNA) missions to chosen smallsat payloads, such as BisonSat, which was the first satellite developed by a tribal college, and STMSat-1, which was the first cubesat built by an elementary school. Programs such as CSLI greatly reduce the resources necessary to send hardware to space, thereby expanding access to space beyond private industry and governmental agencies. As access to space increases, however, so does the need for accurate hardware and software testing capabilities that are equally as accessible.

One such area of testing that is difficult to access is dynamics testing. The dynamics of the space environment are especially difficult to recreate on earth due to the unconstrained nature of rotation and translation enabled by microgravity. Recreating these effects typically requires dedicated simulators that are limited in their accessibility to these smaller institutions. A testing platform that caters to these institutions would need to be inexpensive in its acquisition and straightforward in its construction, with the desired quality of being nonrestrictive in its operation. One such type of platform that fits these desired qualities is multi-rotor vehicles. These vehicles are inexpensive to acquire and are easily constructed; their conventional configurations lack controllability in two of the six degrees of freedom, though there are novel configurations that expand their controllability to all six degrees of freedom, giving them the potential to provide accessible alternatives to traditional spacecraft dynamics simulation platforms. As smaller institutions and organizations continue to develop and launch smallsats, it is vital that spacecraft dynamics simulation capabilities compatible with their resources are developed in order to further increase the quality of the research these institutions conduct; thus, the goal of this thesis is to evaluate the feasibility of 6-DOF controllable multi-rotor vehicles as spacecraft dynamics simulation platforms in

order to increase the accessibility of simulation capabilities for smaller institutions.

1.2 Thesis Contributions

In this thesis, the process of implementing an existing design of a 6-DOF controllable multi-rotor vehicle is begun and its feasibility as a smallsat dynamics simulation platform is evaluated. Progress toward a complete physical implementation is demonstrated, including a CAD model of the structural elements, complete scale designs for 3D printable components, and a price list of chosen hardware components and their quantities. The current state of the vehicle's physical implementation is documented and necessary steps toward its completion are listed. A simulation of the vehicle is developed using the Gazebo dynamics simulator and a software-in-the-loop variant of the Pixhawk flight controller. The controller logic used in the original vehicle's implementation is written into the Pixhawk flight software, and multiple test simulations are conducted to validate the vehicle's performance and determine its feasibility as a smallsat dynamics simulation platform. The end results are detailed plans for a physical implementation of the vehicle and progress toward its construction, a dynamics simulation of the vehicle for testing and validation purposes, and the flight control and trajectory generation software necessary to ensure its operation. Future work steps for both the physical and simulated variants of the vehicle are provided.

1.3 Document Preview

In Chapter 2, a brief review of the literature is conducted in the context of spacecraft dynamics simulators, with emphasis on ascertaining their capabilities of simulating all six degrees of freedom. Next, another brief review adapted from [2] on capable 6-DOF controllable

multi-rotor vehicle designs is given, with designs grouped by the configurability of their rotor arrangements. In Chapter 3, an overview is given of the selected vehicle's design as well as the control logic used in its original implementation. Pre-existing multi-rotor control and communications framework used in this implementation are also discussed. Chapter 4 discusses the work done in this thesis to physically construct the chosen vehicle design, and the steps necessary to complete this effort. Chapter 5 details the work done to implement the flight software and simulation components of the chosen vehicle design. In Chapter 6, the simulated vehicle is tasked with tracking trajectories intended to validate its performance and determine its feasibility as a smallsat dynamics simulation platform, after which results are discussed. Finally, in Chapter 7, concluding remarks and necessary future steps are given.

Chapter 2

Review of Literature

2.1 Spacecraft Dynamics Simulators

When testing hardware and software intended for micro-gravity environments, the most effective test environment is the orbital environment. In conjunction with NASA, DARPA, and Aurora Flight Sciences, the Massachusetts Institute of Technology Space Systems Laboratory developed the Synchronized Position Hold Engage and Reorient Experimental Satellite (SPHERES) system to provide reusable and reliable micro-gravity testing for software and hardware intended for use in space [3]. The SPHERES system, consisting of three micro-satellites residing on the International Space Station (ISS), utilizes twelve carbon dioxide thrusters to maneuver in all six degrees of freedom within the ISS' micro-gravity environment. While it is an effective test platform, the limited amount of experimental time, the regulatory process in order to utilize this system, and the corresponding time and money costs of sending equipment into space make this system difficult to readily access and use without extensive planning. Therefore, many varieties of ground-based space environment simulators have been developed to provide more accessible testing of spacecraft hardware

and software.

One of the most valuable and most difficult aspects of spacecraft to accurately simulate for long periods of time is micro-gravity. Simulators have been developed with this capability, but they are typically constrained by how long they can operate. Constructed in 1966, NASA's Zero Gravity Research Facility is a vacuum chamber drop tower that provides weightlessness to experiments through free-fall [4]. With a height of 142 meters, the maximum period of weightlessness attainable is 5.18 seconds. A private company called the Zero Gravity Corporation provides longer periods of free-fall by selling flights on a modified Boeing 727 that flies in parabolic arcs. This increases the period of weightlessness to 20 - 30 seconds, and opens the possibility of simulating the effects of Martian, Lunar and zero-gravity environments by modifying the profile of the parabola flown [5]. Without employing free-fall, weightlessness can be simulated using neutral buoyancy tanks, like in the Neutral Buoyancy Laboratory at NASA Johnson Space Center [6]. This 202 by 102 by 40 foot pool simulates weightlessness by submersing objects underwater and equalizing their tendencies to sink and float through combinations of weights and flotation devices, thus making them appear weightless. Primarily used to train astronauts for spacewalks, this facility can also be used to test certain space hardware, though the "false" feeling of weightlessness and the presence of water drag reduce the accuracy of more dynamic simulations.

While recreating micro-gravity is an important aspect of testing space equipment, the methods listed above are rather inefficient when it comes to simulating all six degrees of freedom. To achieve greater efficiency, devices such as air-bearing tables can be used. The Air Bearing Floor at NASA Johnson Space Center operates by suspending test equipment on a bed of compressed air blown through a perforated floor, allowing the equipment to glide over the surface without friction. This two-dimensional simulation of translational spacecraft dynamics is useful for testing rendezvous and docking equipment, but lacks rotational degrees of

freedom beyond the yaw dimension [7]. The Orbital Robotic Interaction, On-orbit servicing, and Navigation (ORION) laboratory at the Florida Institute of Technology combines the air bearing floor concept with pan-tilt units to provide rendezvous and proximity operations simulations in all six degrees of freedom, albeit with bounds on achievable orientations in the angular dimensions [8]. At Virginia Tech, researchers have developed an inexpensive simulator specifically for testing Cubesat attitude control systems that utilizes spherical air bearings. The CubeSat Attitude Control Simulator (CSACS) can test payloads up to 300 lbs with 360° of yaw actuation and ±15° of pitch and roll actuation [9].

The concepts explored thus far allow for multiple degrees of freedom to be simulated in controlled environments, but are typically constrained along those degrees of freedom due to their mechanical or structural designs. To alleviate these restrictions, researchers have begun considering multi-rotor vehicles as potential test platforms.

2.2 Multi-rotor Vehicles as Simulation Platforms

2.2.1 Standard Multi-rotor Vehicles

Recently, researchers at Virginia Tech have developed a system for fully distributed multipurpose autonomous flight using unmanned aerial vehicles, namely multi-rotor vehicles. This system utilizes an OptiTrack motion capture system, Robotics Operating System (ROS), Pixhawk flight controllers, and Raspberry Pi companion computers onboard each vehicle, allowing them to know their position and velocity within the experimental space to a high degree of precision [10]. While this system has already been applied to facilitate autonomous multi-agent decision making, current areas of research include simulating the dynamics of singular and distributed small satellite systems. Examples of these areas include rendezvous and

proximity operations, spacecraft inspection, and formation flying dynamics. These research interests have led to the establishment of the SpaceDrones research group for simulating spacecraft dynamics using multi-rotor vehicles.

The SpaceDrones lab environment represents a unique opportunity for simulating small satellite dynamics in an accessible and effective manner through the use of multi-rotor vehicles. Multi-rotor vehicles are accessible technologically thanks to widely available open-source flight controllers and components, and are relatively easy to assemble and operate. Additionally, as demonstrated by the SpaceDrones lab, frameworks like ROS open up the possibility for operating multiple, distributed agents autonomously, which increases the variety of simulatable scenarios. In its current state, the quad-rotor vehicles used in the SpaceDrones lab environment are capable of simulating all three translational degrees of freedom and one rotational degree of freedom, but are unable to recreate the remaining two rotational degrees of freedom due to coupling between their translational and rotational dynamics. Therefore, for this capability to be realized, a multi-rotor vehicle able to independently control all six of its degrees of freedom and able to interface with the SpaceDrones lab environment would be needed.

2.2.2 6-DOF Controllable Multi-rotor Vehicles

In their paper, Brescianini and D'Andrea identify multiple designs for multi-rotor vehicles independently controllable in all six degrees of freedom [2]. Specifically, they determine two categories in which these designs can be classified: vehicles with non-planar rotor configurations where the orientations of the rotor disks are fixed with respect to the vehicle, and vehicles which can adjust the orientation of their rotors to control the direction of their thrust and torque.

The first category contains vehicles such as the design explored in [11]. After performing a kinematic analysis of vehicle thrusts and torques, a design touting three tilted pairs of rotors arranged spherically about their center in order to produce thrust and torque vectors that fully span Euclidean space was proposed. Similar to this is the "Dextrous Hexrotor", a hexacopter vehicle with rotors similarly canted about the radial axis, which uses a force mapping analysis to determine the control inputs necessary to utilize full 6-DOF controllability [12]. Finally, [13] proposes a vehicle where the rotors are radially canted about the vertices of its hexagonal configuration, and designs a geometric controller to arbitrarily command the vehicle's attitude. These first three vehicles were designed with the goal of achieving full 6-DOF controllability, and are not considered optimal with regard to their configuration. The design presented in [14] additionally determines optimal rotor tilt angles tangentially about the vehicle frame to further increase how effectively its control authority spans the thrust and torque space. In [15], a 6-DOF controllable vehicle with an end effector is explored. Rather than using a hexacopter configuration, the vehicle's configuration is determined by solving an optimization problem seeking to minimize the total vehicle volume, resulting in a non-standard vehicle design. A similar optimization approach is taken in [16], with the optimization problem posed as maximizing the magnitude of the vehicle's smallest attainable thrust and torque vectors rather than minimizing volume or weight.

As noted in Brescianini and D'Andrea's paper, the increased thrust and torque authority that these non-planar configurations provide over planar configurations is accompanied by decreased energy efficiency due to larger internal forces [2]. The second category of 6-DOF controllable vehicles, those that can tilt their rotors during flight to re-orient their thrust and torque vectors, mitigate this potential downside through their ability to minimize their internal forces by varying their rotor tilt angles. [17] utilizes this concept in conjunction with a standard quadcopter configuration to design a vehicle with full 6-DOF controllability

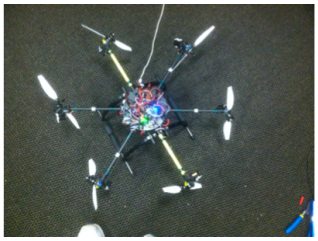
that can reduce its energy expenditure by properly shaping a cost function dependent on the angular velocity of its rotors. [18] also employs this concept, with a focus on quadcopter performance and failure recovery as a result of the increase in controllable degrees of freedom. The design proposed in [19] is unique compared to most variable-angle multi-rotor vehicles; it consists of two coaxial counterrotating ducted propellers for thrust and yaw control and three externally mounted variable-angle ducted propellers for pitch and roll control, together enabling full controllability in all six degrees of freedom.

Brescianini and D'Andrea point out that though these vehicles are able to produce arbitrary forces and torques with reduced energy expenditure, they are typically accompanied by increased weight, complexity, and cost. Additionally, the speed at which thrust and torque responses are executed can be slower due to the time needed to tilt the rotors into the necessary orientations [2]. One design that seeks to combine the benefits of both vehicle categories can be seen in [20], called the Fully-Actuated by Synchronized-Tilting Hexarotor, or "FAST-Hex". This design merges the concept of a hexacopter configuration using symmetrically canted rotors with the concept of variable-tilt rotors, allowing the vehicle to behave as a standard hexacopter when the rotors are not tilted and obtaining full controllability when the propellers are tilted. This vehicle is able to vary its performance from under-actuated to fully-actuated through the addition of a single motor, thus taking advantage of the quickly responsive control authority of fixed-rotor designs and the energy efficient capabilities of tilt-rotor designs.

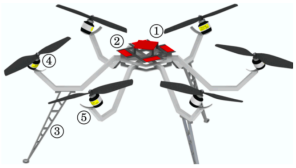
Images of each of the aforementioned vehicle designs can be seen in Figure 2.1. While all of these designs possess the capability to produce thrusts and torques in arbitrary directions, their configurations dictate a preferred orientation in which their rotors are most effective at resisting the effects of gravity. This means that there may be some attitudes where they are not able to hover in place effectively. Brescianini and D'Andrea sought to correct this

(a) Design from [11]

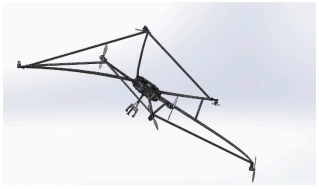
(b) Design from [12]



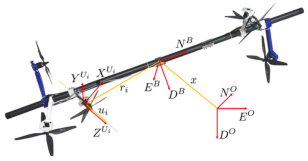
(c) Design from [13]



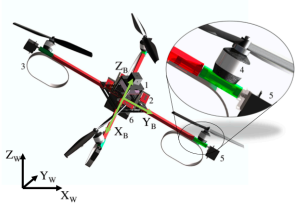
(d) Design from [14]



(e) Design from [15]

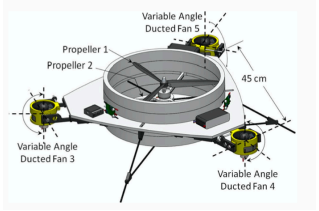


(f) Design from [16]



(g) Design from [17]

(h) Design from [18]



(i) Design from [19]

(j) Design from [20]

Figure 2.1: Images of 6-DOF controllable multi-rotor designs explored in this section, showcasing the similarities and differences in their configurations. All images are adapted from the sources listed in their respective sub-captions.

shortcoming by designing their vehicle to be rotationally invariant, meaning its physical characteristics and thrust/torque performance are independent of its attitude. This makes their vehicle a prime candidate for simulating the dynamics of a small satellite, as small satellites generally have no preferred orientation while in orbit and any simulations of their dynamics would be handicapped by a loss in performance at specific attitudes. For this reason, in addition to its scale-invariant design, Brescianini and D'Andrea's Omnicopter was chosen as the simulation vehicle for the purposes of this thesis.

Chapter 3

Background and Methodology

The following section will explore the technical background for relevant portions of this thesis and describe the various methodologies employed in its execution.

3.1 The Omnicopter

The Omnicopter is a fully-controllable multi-rotor vehicle design proposed by Dario Brescianini and Rafael D'Andrea of the Flying Machine Arena at ETH Zurich [2]. Its ability to independently control its state in all six degrees of freedom allows it to achieve previously unattainable flight maneuvering, which makes it the ideal candidate platform for simulating small-satellite dynamics. This section will give an overview of the design as well as the authors' design process. All information contained in this section originates from the aforementioned authors' work, and is attributed to them as such unless otherwise specified.

3.1.1 Omnicopter Vehicle Design

The Omnicopter's design is the result of an effort to develop a fully actuated multi-rotor vehicle that is able to produce arbitrary thrusts and torques at any attitude. This vastly increases the domain of feasible flight maneuvers when compared to conventional multi-rotor vehicles; multi-rotor vehicles roll, pitch, and yaw in order to translate, while the Omnicopter is able to command any acceleration or rotation independently of its orientation. Because the configuration of the rotors on multi-rotor vehicles dictates the vehicle's performance and behavior, Brescianini and D'Andrea formulated an optimization problem that sought to maximize vehicle agility (which the authors define as the smallest Euclidean norm of the maximum attainable thrust and torque over all directions) while determining the minimum number and optimal orientation of rotors that enable rotationally invariant full actuation, based on static thrust and torque analyses.

For this optimization problem, the smallest magnitude of the maximum attainable thrust and torque over all possible directions was used as the objective function. To be considered rotationally invariant, the vehicle must be able to produce equal thrust and torque in any direction, and must have a symmetric moment of inertia matrix. The first requirement is not achievable with a finite number of motors, but an approximation of it can be achieved through this design by guaranteeing an equal minimum thrust and torque in any direction. To meet the second requirement, assuming that most of the vehicle's moment of inertia is generated by the position of the rotors considered as point masses, the set of possible rotor configurations were restricted to lie on the vertices of regular solids, or combinations of regular solids with coinciding centers. This restriction guarantees that the majority of the vehicle's mass will be symmetrically distributed about its center of gravity, thus minimizing the variance of its moment of inertia when rotating. Because a minimum of six rotors

are needed to attain full actuation in a fixed-rotor configuration, this restriction further requires that the regular solids have at least six vertices along which the rotors are placed. Additionally, because all regular solids have vertices that are equidistant from their centers, the rotors were chosen to lie on the unit sphere such that $\|\boldsymbol{p_i}\|_2 = 1$, where $\boldsymbol{p_i}$ is the position of the ith rotor relative to the vehicle's center of mass. This ensures that the torques are normalized, and that a single unit of rotor thrust results in at most one unit of total thrust and one unit of total torque, which allows the vehicle design to be scale invariant.

The formal optimization problem that was solved to determine the vehicle configuration is:

$$\begin{aligned} \underset{\boldsymbol{P},\boldsymbol{N}}{\text{maximize}} \quad & \arg\max_{\boldsymbol{r}}\{r : \{\boldsymbol{\nu} \in \mathbb{R}^6 |\ \|\boldsymbol{\nu}\|_2 \leq r\} \subseteq \mathcal{V}\} \\ \text{subject to} \quad & \|\boldsymbol{n}_i\|_2 = 1, \forall i \in \{1, \ldots, N\},\ P \in \mathcal{P},\ \sigma_{max}(\widetilde{\boldsymbol{B}}) = \sigma_{min}(\widetilde{\boldsymbol{B}}) \end{aligned} \tag{3.1}$$

where $\sigma_{max}(\widetilde{\boldsymbol{B}})$ and $\sigma_{min}(\widetilde{\boldsymbol{B}})$ are the minimum and maximum singular values of the thrust and torque map $\widetilde{\boldsymbol{B}}$, which is defined as follows:

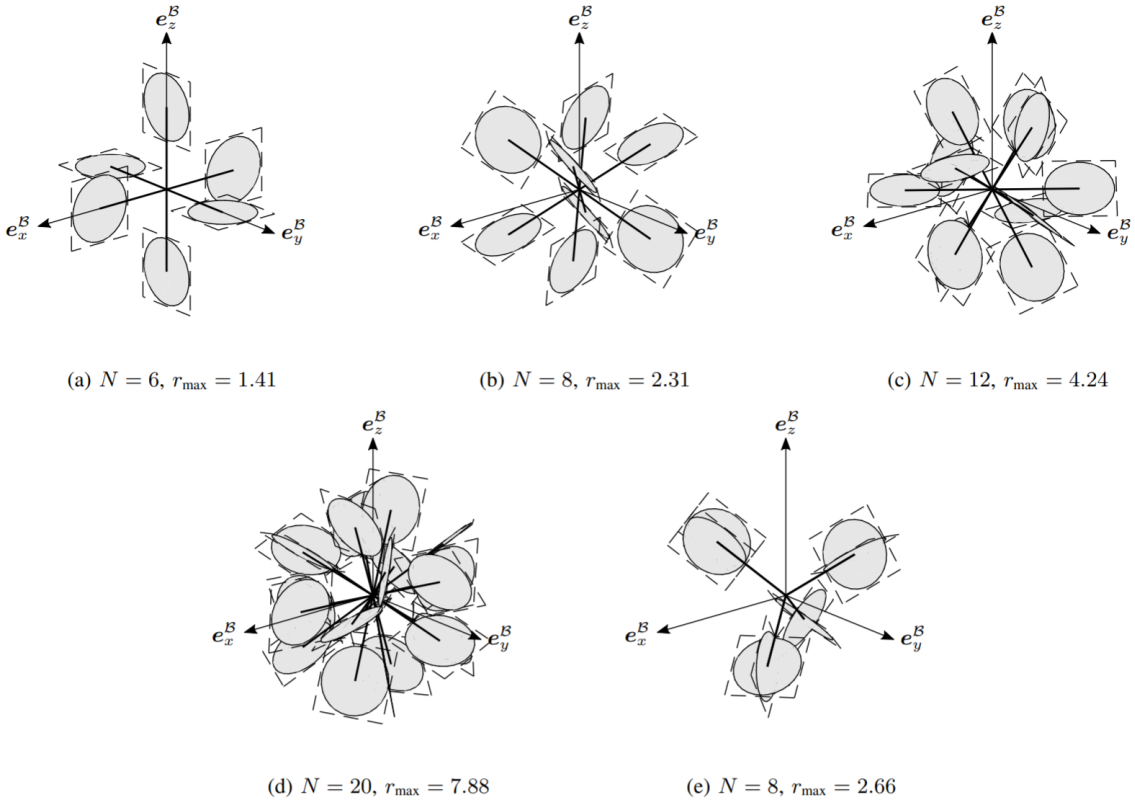
$$\widetilde{\boldsymbol{B}} = \begin{bmatrix} \boldsymbol{N} \\ \boldsymbol{P} \times \boldsymbol{N} \end{bmatrix} \tag{3.2}$$

Essentially, Brescianini and D'Andrea sought to maximize the number and the radial distance of each rotor relative to the center of mass such that the smallest thrust and torque are maximized. This objective is subject to the constraints above, namely that the largest and smallest singular values of the thrust and torque map $\widetilde{\boldsymbol{B}}$, which maps individual rotor

thrusts to the vehicle's forces and torques $\boldsymbol{\nu_{cmd}} = (\boldsymbol{f}_{cmd}, \boldsymbol{t}_{cmd})$ via $\boldsymbol{\nu_{cmd}} = \widetilde{\boldsymbol{B}}\boldsymbol{f}_{rotor,cmd}$, are equal. This guarantees the minimum thrust and torque generated in every direction to be equal. The authors point out that in this design objective the total thrust and torque are weighted equally, though it may seem beneficial to weight one over the other. Despite this, after solving the optimization problem, they determined numerically that the optimal rotor configuration (consisting of the matrices of rotor normal vectors $\boldsymbol{N}$ and position vectors $\boldsymbol{P}$) is independent of the weighting on the thrust and torque, which is equivalent to say that it is independent of the size of $\|\boldsymbol{p_i}\|$. This means that the distance of the rotors from the center of mass can be chosen dependent on torque requirements and is not a contributing factor to the optimal rotor configuration, though it is important to note that the magnitude of attainable torque is inversely related to the vehicle's agility, due to the increase in moment of inertia caused by moving the rotors away from the vehicle's center of mass.

After formulating the aforementioned optimization problem, the authors used MATLAB's constrained optimization function *fmincon* to solve for optimal rotor configurations, which are shown in Figure 3.1a-e for varying number of rotors N.

These N values represent configurations where the motors are constrained to an octahedron ($N = 6$), a cube ($N = 8$), an icosahedron ($N = 12$), and a dodecahedron ($N = 20$). The authors note that for some values of $N > 6$, multiple rotor configurations exist that provide rotationally invariant performance. Typically, these involved rotor pairs coinciding at vertices of regular solids with rotor normals aligned perpendicularly to each other. This resulted in higher attainable thrust and torques, but these configurations were not considered due to the physically impractical intersection of their rotor disks. Thus, these unrealistic optimal solutions were discarded in favor of suboptimal solutions that were more practical to construct. One of these discarded solutions is shown in of Figure 3.1e. In each of the generated solutions, the normal vectors of each rotor disk are perpendicularly aligned to their position



(a) $N = 6$, $r_{\max} = 1.41$ (b) $N = 8$, $r_{\max} = 2.31$ (c) $N = 12$, $r_{\max} = 4.24$

(d) $N = 20$, $r_{\max} = 7.88$ (e) $N = 8$, $r_{\max} = 2.66$

Figure 3.1: Omnicopter optimization results, adapted from [2]. The optimal rotor configurations are shown for varying number of rotors, N. r_{max} denotes the radius of the largest Euclidean ball centered at the origin that is fully enclosed by the given design's set of attainable thrust and torques.

vectors to maximize their torque output. The author also notes that all of the configurations generated achieved the maximum possible singular values for their thrust and torque maps, meaning that each one is capable of producing the largest possible thrust and torque output.

An important note is made regarding the minimum rotor thrust magnitude: because the vehicle cannot create thrusts of arbitrarily small magnitude, a vehicle design with only six rotors would encounter orientations where no thrust and torque can be produced, jeopardizing the vehicle's ability to hover at any arbitrary attitude. To overcome this, the vehicle must use a configuration with more than six rotors. This over-actuation fills in the disconnected

portions of the thrust and torque map caused by the finite lower bound on the minimum attainable thrust. Because of this, the optimal ($N = 6$) solution was not chosen for the vehicle's design. The next most optimal (and physically feasible) solution was chosen with ($N = 8$), representing a cube with rotors located at the vertices. The matrices $\boldsymbol{P}$ and $\boldsymbol{N}$, containing the position vectors and normal vectors for each rotor, are

$$\boldsymbol{P} = \frac{1}{\sqrt{3}} \begin{bmatrix} 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 & 1 & 1 & -1 & -1 \\ 1 & 1 & 1 & 1 & -1 & -1 & -1 & -1 \end{bmatrix} \tag{3.3}$$

$$\boldsymbol{N} = \begin{bmatrix} -a & b & -b & a & a & -b & b & -a \\ b & a & -a & -b & -b & -a & a & b \\ c & -c & -c & c & c & -c & -c & c \end{bmatrix} \tag{3.4}$$

where $a = \frac{1}{2} + \frac{1}{\sqrt{12}}$, $b = \frac{1}{2} - \frac{1}{\sqrt{12}}$, and $c = \frac{1}{\sqrt{3}}$. A visual representation of this design can be seen in Figure 3.1b, and the fully constructed vehicle is shown in Figure 3.2.

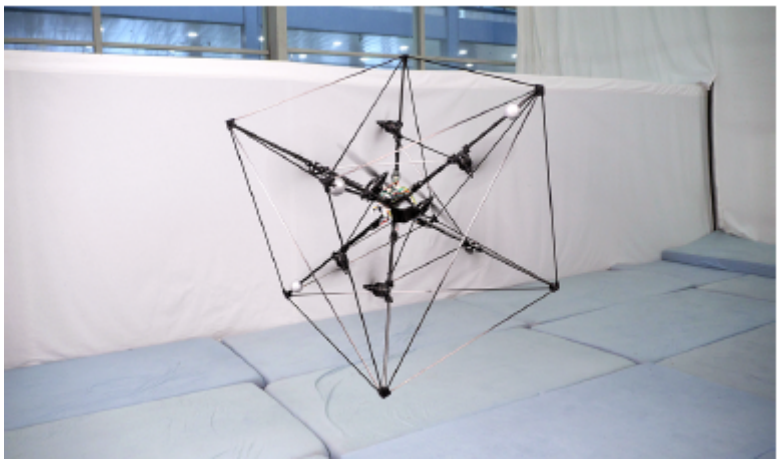


Figure 3.2: Omnicopter vehicle, designed by Brescianini and D'Andrea, fully assembled. Adapted from [2].

3.1.2 Omnicopter Control Allocation & Design

The Omnicopter's control scheme was designed such that the vehicle would be able to track arbitrary position and attitude trajectories $\boldsymbol{p_{des}}(t)$ and $\boldsymbol{q_{des}}(t)$. A cascaded control architecture is implemented, with assumptions made that the controller models are able to track set point changes without delay. Figure 3.3 shows a diagram of the Omnicopter's control architecture.

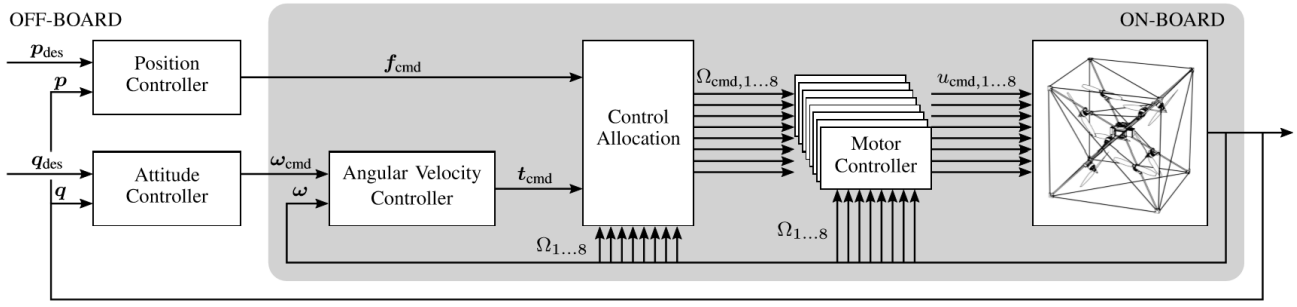


Figure 3.3: Omnicopter control architecture. The position and attitude controllers run separately due to the decoupling of the vehicle's translational and rotational dynamics. Adapted from [2]

.

Position Control

The authors model the position controller as a 2nd-order dynamic system with time constant τ_{pos} and damping ratio ζ_{pos}. The general form of the controller is

$$\ddot{\boldsymbol{p}} = \frac{1}{\tau_{pos}^2}\boldsymbol{p_{err}} + \frac{2\zeta_{pos}}{\tau_{pos}}\dot{\boldsymbol{p}}_{\boldsymbol{err}} + \ddot{\boldsymbol{p}}_{\boldsymbol{des}} \tag{3.5}$$

where $\boldsymbol{p_{err}} = \boldsymbol{p_{des}} - \boldsymbol{p}$ is the error between the desired position and the current position, with the current position measured using a motion capture imaging system. After evaluating

the above control law with respect to the vehicle's translational dynamics model, the force $\boldsymbol{f_{cmd}}$ necessary to track $\boldsymbol{p_{des}}$ is

$$\boldsymbol{f_{cmd}} = m\boldsymbol{R}(\boldsymbol{q})\left(k_{pos,p}\boldsymbol{p_{err}} + k_{pos,i}\int \boldsymbol{p_{err}}dt + k_{pos,d}\dot{\boldsymbol{p}}_{\boldsymbol{err}} + \ddot{\boldsymbol{p}}_{\boldsymbol{des}} + \boldsymbol{g}\right) \tag{3.6}$$

where subscripts p, i, and d denote position, integral, and derivative gains, m is the vehicle mass, $\boldsymbol{R}(\boldsymbol{q})$ is the rotation from the inertial frame to the vehicle body frame dependent on the current attitude quaternion $\boldsymbol{q}$, and $\boldsymbol{g} = (0, 0, 9.81)$ m/s^2 is the gravitational acceleration vector. In the above control law, an integral term is included with time constant $\tau_{pos,i}$ to eliminate steady-state error and increase tracking accuracy. The proportional, integral, and derivative gains are given by

$$k_{pos,p} = \frac{1}{\tau_{pos}^2} + \frac{2\zeta_{pos}}{\tau_{pos}\tau_{pos,i}} \tag{3.7}$$

$$k_{pos,i} = \frac{1}{\tau_{pos}^2\tau_{pos,i}} \tag{3.8}$$

$$k_{pos,d} = \frac{2\zeta_{pos}}{\tau_{pos}} + \frac{1}{\tau_{pos,i}} \tag{3.9}$$

Attitude Control

The attitude controller is designed to be asymptotically stable over the range of possible attitudes, allowing any desired attitude to be commanded. The vehicle's attitude is represented by a unit quaternion, $\boldsymbol{q} = (q_0, \widetilde{\boldsymbol{q}})$, where q_0 is the scalar portion and $\widetilde{\boldsymbol{q}}$ is the vector portion. The attitude error quaternion $\boldsymbol{q_{err}}$ is

$$\boldsymbol{q_{err}} = \boldsymbol{q_{des}} \cdot \boldsymbol{q} \tag{3.10}$$

where in this context, $\cdot$ denotes quaternion multiplication, and the current attitude quaternion $\boldsymbol{q}$ is measured using a motion capture imaging system. The attitude kinematics associated with this error are

$$\dot{\boldsymbol{q}}_{\boldsymbol{err}} = \frac{1}{2}\begin{bmatrix} 0 \\ \boldsymbol{\omega}_{des} \end{bmatrix} \cdot \boldsymbol{q_{err}} - \frac{1}{2}\boldsymbol{q_{err}} \cdot \begin{bmatrix} 0 \\ \boldsymbol{\omega} \end{bmatrix} \tag{3.11}$$

where $\boldsymbol{\omega}_{des}$ is the angular velocity that satisfies the attitude kinematics for the given trajectory, and $\boldsymbol{\omega}$ is the vehicle's current angular velocity. The control law that globally stabilizes the vehicle's attitude around the equilibria $\boldsymbol{q_{err}} = (\pm 1, 0, 0, 0)$, with time constant τ_{att}, is

$$\boldsymbol{\omega_{cmd}} = \frac{2}{\tau_{att}} \text{sgn}(q_{err,0}) \widetilde{\boldsymbol{q}} \tag{3.12}$$

To eliminate steady-state errors and allow desired trajectories beyond the equilibria to be tracked, an integral term with time constant $\tau_{att,i}$ and a feed-forward term are added, leading to the final attitude control law

$$\boldsymbol{\omega_{cmd}} = k_{att,p} \text{sgn}(q_{err,0}) \widetilde{\boldsymbol{q}} + k_{att,i} \int \text{sgn}(q_{err,0}) \widetilde{\boldsymbol{q}} dt + \boldsymbol{R}(\boldsymbol{q_{err}})^{-1} \boldsymbol{\omega_{des}} \tag{3.13}$$

with control gains

$$k_{att,p} = \frac{2}{\tau_{att}} + \frac{2}{\tau_{att,i}}, \quad k_{att,i} = \frac{2}{\tau_{att}\tau_{att,i}} \tag{3.14}$$

Angular Velocity Control

The commanded angular velocity is then tracked by an inner angular velocity control loop, which the authors model to resemble a linearized first order system with time constant τ_ω. This controller determines the body torque necessary to track the desired attitude trajectory through the commanded angular velocity vector:

$$\boldsymbol{t_{cmd}} = \frac{1}{\tau_\omega}\boldsymbol{J}(\boldsymbol{\omega_{cmd}} - \boldsymbol{\omega}) + \boldsymbol{\omega} \times \left(\boldsymbol{J\omega} + \sum_{i=1}^{8} J_{rotor}\Omega_i \boldsymbol{n}_i \right) \tag{3.15}$$

where $\boldsymbol{J}$ is the vehicle's moment of inertia matrix, Ω_i is the i^{th} rotor's angular velocity, $\boldsymbol{n}_i$ is the corresponding rotor's normal vector, and $\boldsymbol{\omega}$ is the current vehicle angular velocity, measured using an onboard inertial measurement unit. J_{rotor} and Ω_i are the moments of inertia and angular velocities of each rotor, respectively.

Control Allocation

With the forces and torques $\boldsymbol{\nu_{cmd}} = (\boldsymbol{f}_{cmd}, \boldsymbol{t}_{cmd})$ needed to track the desired position and attitude trajectories computed, a strategy for distributing commands to the actuators is required. It was mentioned previously that the vehicle's forces and torques can be related to the individual rotor forces and torques through the force and torque map $\widetilde{\boldsymbol{B}}$. Going forward, the authors modify this matrix into $\boldsymbol{B}$ by taking into account rotor thrust and drag torque:

$$\boldsymbol{B} = \begin{bmatrix} \boldsymbol{N} \\ \boldsymbol{P} \times \boldsymbol{N} + \boldsymbol{N}\boldsymbol{K} \end{bmatrix} \tag{3.16}$$

where the matrix $\boldsymbol{K}$ is

$$\boldsymbol{K} = \kappa \boldsymbol{I}_{8X8} \tag{3.17}$$

with κ as the rotor thrust-to-drag coefficient.

The relationship between the vehicle's forces and torques and the individual rotor forces and torques is given by

$$\boldsymbol{\nu} = \boldsymbol{B}\boldsymbol{f}_{\boldsymbol{rotor}} \tag{3.18}$$

Using the right pseudo-inverse of $\boldsymbol{B}$, $\boldsymbol{B}^{\dagger} = \boldsymbol{B}^{T}(\boldsymbol{B}\boldsymbol{B}^{T})^{-1}$, the previous equation can be rearranged to solve for the individual rotor forces needed to produce the desired vehicle forces and torques:

$$\hat{\boldsymbol{f}}_{\boldsymbol{rotor}} = \boldsymbol{B}^{\dagger}\boldsymbol{\nu}_{\boldsymbol{cmd}} \tag{3.19}$$

This is the foundational concept behind how the Omnicopter's control allocation scheme

functions. Due to the Omnicopter's overactuation, however, the set of individual rotor thrusts that produces the commanded vehicle forces and torques is not unique. This is because the null space of the force and torque map $\boldsymbol{B}$ is non-trivial. To account for this, the authors further pose the control allocation problem as an optimization problem. The optimization problem has two goals, one short term and one long term. The short term goal is to minimize the difference in present and commanded rotor thrusts, which aims to account for some dynamics of the motors that are assumed to track instantly in the control loops. By minimizing the difference between the present and desired thrusts, the amount of delay introduced while the motor reaches the desired thrust is minimized as well. This goal is achieved by minimizing the square of the difference in desired and present thrust, as given by the following mathematical relation:

$$\sum_{i=1}^{8}(f_{rotor,cmd,i} - f_{rotor,i})^2 \tag{3.20}$$

The long term goal is to reduce the vehicle's overall power consumption, thereby increasing its operational efficiency. This is achieved by minimizing the cubed square root of the commanded rotor thrust, which is proven to be sufficient in Appendix B of [2]:

$$\sum_{i=1}^{8}|f_{rotor,cmd,i}|^{3/2} \tag{3.21}$$

The authors note that the short term objective is insufficient in making up for the motor's inability to track commanded setpoints without delay, as was its purpose. Therefore, an additional hysteresis constraint is added that prevents solutions to the optimization problem

that requires rotors to change direction which have already done so within a specified time period. Thus, the formal optimization problem can be stated as follows:

$$\begin{aligned} \underset{\boldsymbol{f}_{rotor,cmd}}{\text{minimize}} \quad & \sum_{i=1}^{8}(1-\epsilon)|f_{rotor,cmd,i}|^{3/2} + \epsilon(f_{rotor,cmd,i} - f_{rotor,i})^2 \\ \text{subject to} \quad & \boldsymbol{\nu}_{cmd} = \boldsymbol{B}\boldsymbol{f}_{rotor,cmd}, \\ & \boldsymbol{f}_{rotor,cmd} \in \mathcal{F}_{rotor}^{hyst} \end{aligned} \tag{3.22}$$

where ϵ is a weighting term between the short and long term objectives, and $\mathcal{F}_{rotor}^{hyst}$ is the set of rotor thrusts that does not require a direction reversal within a specified period of time τ_{hyst}.

The aforementioned optimization problem requires solutions to be computed for each of the eight rotors in real time, which is not feasible on typical microcontroller architectures. Therefore, to solve the above optimization problem on a microcontroller in real time, the authors utilize the null space of the thrust and torque map $\boldsymbol{B}$ to simplify the optimization problem. Equation 3.19, relating vehicle forces and torques to individual rotor forces, is modified by the addition of two scalar biases ϕ_1 and ϕ_2 multiplying the vectors that span the null-space of $\boldsymbol{B}$, $\boldsymbol{\eta_1}$ and $\boldsymbol{\eta_2}$:

$$\boldsymbol{f}_{rotor,cmd} = \boldsymbol{B}^{\dagger}\boldsymbol{\nu}_{cmd} + \phi_1\boldsymbol{\eta}_1 + \phi_2\boldsymbol{\eta}_2 \tag{3.23}$$

with $\boldsymbol{\eta}_1 = (1,1,1,1,0,0,0,0)$ and $\boldsymbol{\eta}_2 = (0,0,0,0,1,1,1,1)$. Adding the biases multiplying the null space vectors to the above equation does not affect the total force and torque outputs

of the vehicle due to their contributions to the rotor thrusts lying within the nullspace of $\boldsymbol{B}$. Therefore, the rotor thrust values can be modified to achieve the short and long term optimization goals without affecting the vehicle's performance. The optimization problem is reduced in complexity to simply finding the optimal scalar biases, ϕ_1 and ϕ_2, that minimize the goals mentioned previously. To maintain feasibility of the resulting commanded motor thrusts, the biases ϕ_1 and ϕ_2 are constrained to the sets

$$\Phi_1 = \{\phi_1 \in \mathbb{R} | f_{min}\mathbf{1} \preceq |\boldsymbol{P}_1(\boldsymbol{B}^\dagger \boldsymbol{\nu}_{cmd} + \phi_1 \boldsymbol{\eta}_1)| \preceq f_{max}\mathbf{1}\} \tag{3.24}$$

$$\Phi_2 = \{\phi_2 \in \mathbb{R} | f_{min}\mathbf{1} \preceq |\boldsymbol{P}_2(\boldsymbol{B}^\dagger \boldsymbol{\nu}_{cmd} + \phi_2 \boldsymbol{\eta}_2)| \preceq f_{max}\mathbf{1}\} \tag{3.25}$$

where $\boldsymbol{P}_1 = (\boldsymbol{I}_{4x4}, \mathbf{0}_{4x4})$, $\boldsymbol{P}_2 = (\mathbf{0}_{4x4}, \boldsymbol{I}_{4x4})$, and f_{min} and f_{max} are the minimum and maximum achievable rotor thrusts, respectively. Equation 3.23 can be inserted into Equation 3.22, thereby recasting the optimization problem from one solving for eight individual rotor commands to one solving for the two scalar biases. The optimization problem for ϕ_1, which addresses motors 1 through 4, is shown below:

$$\begin{aligned} \underset{\phi_1}{\text{minimize}} \quad & \sum_{i=1}^{4} (1-\epsilon)|\hat{f}_{rotor,cmd,i} + \phi_1|^{3/2} + \epsilon(\hat{f}_{rotor,cmd,i} + \phi_1 - f_{rotor,i})^2 \\ \text{subject to} \quad & \phi_1 \in \Phi_{1,hyst} \end{aligned} \tag{3.26}$$

where $\hat{f}_{rotor,cmd} = \boldsymbol{B}^\dagger \boldsymbol{\nu}_{cmd}$ and $\Phi_{1,hyst}$ is analogous to $\mathcal{F}_{rotor}^{hyst}$. Substituting ϕ_1 with ϕ_2 and summing instead over rotors 5 through 8 gives the objective function for ϕ_2. The authors note that the convex nature of the objective function allows for the optimal biases ϕ_1^* and

ϕ_2^* to be found by first solving for the unconstrained biases $\bar{\phi}_1^*$ and $\bar{\phi}_2^*$, and then evaluating whether or not they belong to the sets $\Phi_{1,hyst}$ and $\Phi_{2,hyst}$. If they do, then $\phi_1^* = \bar{\phi}_1^*$ and $\phi_2^* = \bar{\phi}_2^*$. If not, then the optimal biases are the projections of the unconstrained biases onto the boundaries of the constraining sets.

To solve the above objective functions in real-time onboard the vehicle's microcontroller, the derivatives of each objective function were evaluated with respect to the biases and set equal to zero. The biases are then found by solving the rootfinding problem using the bisection method, which is achievable in real time on a low-cost microcontroller. The derivative of the objective function for ϕ_1 is shown below:

$$\sum_{i=1}^{4} \frac{3}{2}(1-\epsilon)|\hat{f}_{rotor,cmd,i} + \phi_1|^{1/2} + 2\epsilon(\hat{f}_{rotor,cmd,i} + \phi_1 - f_{rotor,i}) = 0 \tag{3.27}$$

The derivative for ϕ_2 can be found analogously. Once the optimal biases are found, the individual rotor thrust commands can be found using Equation 3.23 and can then be converted to individual rotor angular velocities using the following relation from momentum-blade element theory and the rotor coefficient of thrust c_f, which has units of $\frac{N}{(rad^2/s^2)}$. [21]:

$$\Omega_{cmd,i} = \text{sgn}(f_{rotor,cmd,i})\sqrt{\frac{|f_{rotor,cmd,i}|}{c_f}} \tag{3.28}$$

Motor Control

With these individual rotor angular velocities in hand, the final piece of the control architecture is the motor control. The authors note that in a traditional multi-rotor vehicle, the

total thrust is independent of the individual motors' angular velocities, due to the alignment of their rotors in a single plane. For the Omnicopter, the non-planar rotor configuration results in the total thrust and torque produced being very much a function of each rotors' angular velocity. Thus, a motor angular velocity controller is constructed in the manner of a first-order system with time constant τ_{mot}, with the commanded output being the voltage input to each motor, $u_{cmd,i}$:

$$u_{cmd,i} = \frac{RJ_{rotor}}{k_e}\left(k_{mot,p}(\Omega_{cmd,i} - \Omega_i) + k_{mot,i}\int(\Omega_{cmd,i} - \Omega_i)dt\right) + \bar{u}(\Omega_i) \tag{3.29}$$

where R is the resistance, J_{rotor} is the rotor's moment of inertia about the axis perpendicular to its rotation, k_e is the motor constant,$\bar{u}(\Omega_i)$ is an experimentally determined relationship between voltage and angular velocity, and the control gains are given by

$$k_{mot,p} = \frac{\tau_{mot,int}}{\tau_{mot}(\tau_{mot,int} - \tau_{mot})} \tag{3.30}$$

$$k_{mot,p} = \frac{1}{\tau_{mot}(\tau_{mot,int} - \tau_{mot})} \tag{3.31}$$

Controller Parameters

This concludes the discussion of relevent control algorithms used by the Omnicopter. All of the algorithms discussed will be implemented into the recreation of the Omnicopter developed in this work to ensure it reliably tracks position and attitude setpoints. As this is a recreation, the control algorithms and controller gains will not be modified prior to implementation. Modifying these values may become possible in the future as familiarity with the vehicle

increases. Table 3.1, using data from [2], lists all relevant constants and the values needed to calculate the controller gains used in each algorithm listed previously.

Table 3.1: Controller parameters and time constants. Data from [2].

Parameter:	Description:	Value
τ_{pos}	Position control time constant	0.325 s
$\tau_{pos,i}$	Position control integral time constant	1.33 s
ζ_{pos}	Position control damping ratio	1.0
τ_{att}	Attitude control time constantt	0.39 s
$\tau_{att,i}$	Attitude control integral time constant	1.5 s
τ_{ω}	Angular velocity control time constant	0.044 s
ϵ	Control allocation objective weight	0.95
τ_{hyst}	Rotation direction temporal hysteresis	0.75 s
τ_{mot}	Motor control time constant	0.032 s
$\tau_{mot,int}$	Motor control integral time constant	0.199 s

3.2 Multi-rotor Simulation and Control Frameworks

In this implementation of the Omnicopter, pre-existing simulation and control frameworks were used to facilitate the integration of vehicle components developed in this thesis. These include Gazebo, ROS, MAVLink, and the Pixhawk flight controller architecture.

3.2.1 Gazebo Simulator

To simulate the dynamics of the Omnicopter, the Gazebo robotics simulator was used. Gazebo is a high-fidelity open-source physics simulator that provides customizable dynamic and physical environments, allowing various scenarios to be simulated. This enables the acceleration of algorithmic development, controller design, system validation, and the progression of learning-based systems [22]. Gazebo uses the Simulation Discription Format (SDF) to define models, meaning it is extremely flexible with regards to custom model

development.

Each Gazebo model is fully defined by a .sdf file, which contains information on the model's links and joints. Links are individual elements of the model that are simulated physically. The mass, inertia matrix, visual, and collision properties of each link are defined within the .sdf file so that each link can interact dynamically with the environment. Examples of links can include the individual segments of a robotic arm, or the individual rotors and chassis on a multi-rotor vehicle. Typically, the motion of the links relative to one another are constrained using joints. Multiple types of joints exist, including "revolute" joints that act as hinges, "prismatic" joints that allow links to translate linearly within a specified range, "ball" joints that act as ball and sockets, and "piston" joints that are similar to prismatic joints but allow rotation. An example of a joint would be a "revolute" joint between a rotor and the chassis of a multi-rotor vehicle, which would physically constrain it to the chassis but allow it to rotate freely about a defined axis (e.g., its rotor normal). Model joints can have their velocities driven by internal or external plugins, allowing motorized or actuated components to be simulated. Thus, by carefully constructing relevant links and joints, a large variety of actuated systems can be constructed as models in Gazebo. Once constructed, models are simulated utilizing open-source physics engines designed to provide a realistic dynamics environment for simulating multi-rotor vehicles.

While Gazebo provides a high-fidelity environment for the models within the simulation, its plugin-friendly architecture allows external systems to interact with the models. By allowing every physical aspect of the model to be accessed and modified by plugins, external controllers and algorithms developed for the systems being simulated can be tested in real-time, which software-in-the-loop testing and even hardware-in-the-loop tests to be run on target microcontrollers. Coupling this with its ability to interface with ROS makes Gazebo an invaluable tool for simulating hardware- and software-based robotics tools before interacting

with a physical testing environment.

3.2.2 ROS

The Robot Operating System (ROS), while not a true operating system in the computational sense, is a framework of open-source tools and methods that seek to facilitate the development of complex robotic behaviors and technology by providing researchers with a suite of standardized tools and conventions [23].

ROS operates by hosting a publisher/subscriber based messaging service. When a ROS Master server is established, clients connected to the server (known as nodes) can broadcast data to individual message boxes, called topics, or subscribe to a topic to receive any data being broadcasted to it. ROS nodes are executable files that utilize the ROS framework to communicate with other nodes connected to the ROS Master, meaning that the scope of a robotics implementation in ROS can range from sensors on a single vehicle exchanging information to multiple vehicles in the same space sharing pose information. ROS Topics are defined by the messages they accept, which can contain multiple fields of different data types, and can be user-defined. The modularity provided by abstracting information sharing between nodes is part of what makes ROS a useful tool for robotics development.

ROS' inherent modularity is further affirmed by its support of packages, which are pre-configured software distributions that provide specific capabilities. Packages can be installed and uninstalled independently of each other, allowing tools relating to motion planning, computer vision, communications protocols, and more, to be accessed with ease by ROS users. The modularity that ROS provides enables researchers to focus more on their own specific development goals and allows the challenge of integrating all of the necessary components to be handled by ROS.

3.2.3 Pixhawk

To facilitate integration of the flight hardware and software components, a Pixhawk Flight Controller Unit (FCU) running the PX4 firmware was chosen as the main flight computer. PX4 is an open-source flight controller code base that emphasizes modularity in hardware and software components, configurability in user integration, and open-source collaboration between developers [24]. Originally developed as a platform for computer vision research in 2009 by a team of students at ETH Zurich, continued development and open-source contributions enabled the PX4 firmware to become an industry standard in multi-rotor and RC vehicle controls. PX4, based on the NuttX real-time operating system, is designed around a core set of software modules, including input-output control, peripheral communications, the uORB internal publish-subscribe based messaging system, and navigation components. Additional features can be added by the end user through custom applications that are run as regularly scheduled tasks within the NuttX environment without adversely affecting the overall performance of the core system. This modular capability makes PX4 an exceptionally useful tool in the development of flight controller software.

Figure 3.4 shows a block diagram overview of the PX4 architecture. Each of the major modules exchange data via the publish/subscribe based uORB messaging system, though some directly access parameter values within the flight controller memory. The typical command flow begins with normalized thrust, roll, pitch, and yaw setpoints originating from manual radio control inputs, autonomous command inputs from onboard controller modules, or external offboard commands being sent through the Micro Air Vehicle Link messaging protocol, or MAVLink. Once the commands have been routed correctly through the `commander` state machine, they are sent to the onboard airframe mixer to be transformed from thrust, yaw, pitch, and roll commands normalized between -1 and $+1$ to distinct motor

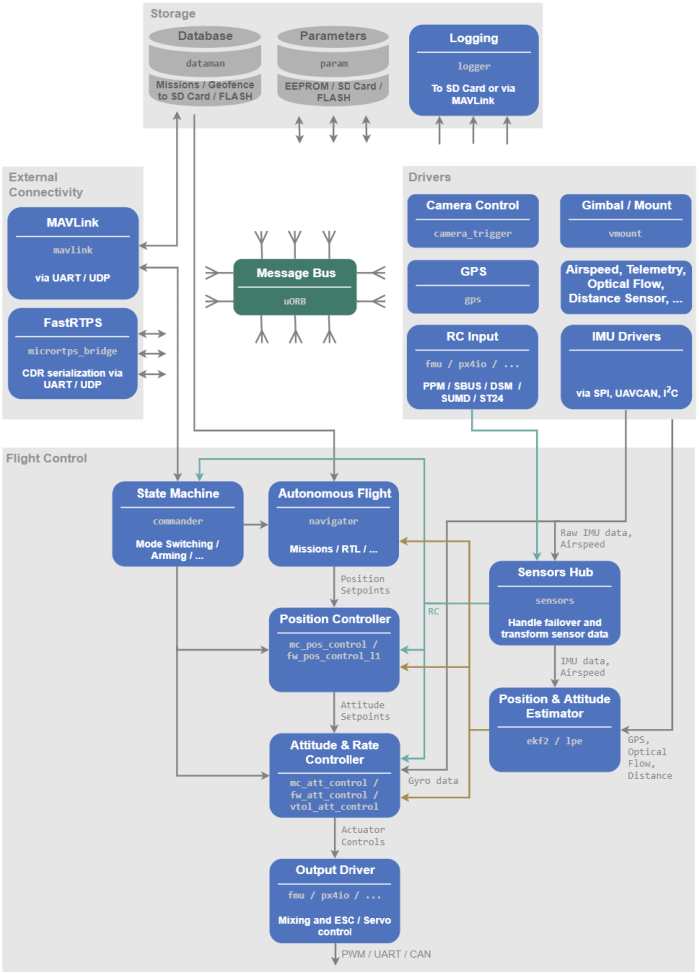


Figure 3.4: PX4 architecture block diagram, showing the major modules and their interactions. Adapted from [24].

actuation signals.

Airframe mixers are PX4's method for introducing a layer of abstraction between controller outputs and particular airframe configurations. These mixers can be thought of as matrices that transform the normalized command values sent by the command source into individual actuator commands that are delivered to each motor onboard, as represented by Equation 3.32:

$$\begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \end{bmatrix} = \begin{bmatrix} M_{1T} & M_{1R} & M_{1P} & M_{1Y} \\ M_{2T} & M_{2R} & M_{2P} & M_{2Y} \\ M_{3T} & M_{3R} & M_{3P} & M_{3Y} \\ M_{4T} & M_{4R} & M_{4P} & M_{4Y} \end{bmatrix} \begin{bmatrix} \boldsymbol{T}_{cmd} \\ \boldsymbol{R}_{cmd} \\ \boldsymbol{P}_{cmd} \\ \boldsymbol{Y}_{cmd} \end{bmatrix} \tag{3.32}$$

where $\boldsymbol{T_{cmd}}$, $\boldsymbol{R_{cmd}}$, $\boldsymbol{P_{cmd}}$, and $\boldsymbol{Y_{cmd}}$ are normalized thrust, roll, pitch, and yaw commands, respectively. In this idealized representation, the normalized thrust, roll, pitch, and yaw commands are multiplied by the mixer matrix coefficients to produce the individual actuator outputs u_1 through u_4. In general, the structure of a mixer and the coefficients within are dependent on the configuration of the vehicle's airframe, and must be correctly constructed so that the command outputs for thrust are appropriately allocated to motors that contribute to thrust, the command outputs for yaw are appropriately allocated to motors that contribute to yaw, etcetera. In this example, the vehicle using this mixer would have four actuators, as the mixer has four outputs. In the PX4 source code, mixer files are defined as follows:

```
M: 2
O:      10000  10000      0 -10000  10000
S: 0 0  -6000  -6000      0 -10000  10000
S: 0 1   6500   6500      0 -10000  10000
```

Source Code 1: PX4 Mixer Code Example

The above mixer code is sufficient to describe the input/output relationship between two command inputs and a single actuator output. The first line, `M: <control count>`, begins a block of code corresponding to a single actuator and denotes that it depends on `<control count>` input commands. In this case, the output being defined depends on two input commands. The next line beginning with `O:` handles scaling and offsets applied to the

output signal before it's sent to the actuator. The syntax is:

```
O: <-ve scale> <+ve scale> <offset> <lower limit> <upper limit>
```

It is important to note that all values are scaled by a factor of 10000; thus, in the example shown, output values are unchanged in both the positive and negative ranges, there are no offsets, and the output values can occupy the full range of -1 to $+1$. The next `<control count>` lines, beginning with `S:`, perform the same operations as the `O:` line but on the incoming command control signals rather than the actuator output. The syntax for these lines is

```
S: <group> <index> <-ve scale> <+ve scale> <offset> <lower limit> <upper limit>
```

which is identical to the second line of the example mixer except for the addition of two preceding values. The `<group>` denotes which actuator control group this input command value originates from, and the `<index>` value denotes the corresponding control channel within the actuator control group. In this example, the first input comes from the roll channel (channel 0) within the flight control group (`actuator_controls_0`), and it is scaled by -0.6 in both the positive and negative ranges. The second input comes from the pitch channel (channel 1) within the same control group, and is scaled by 0.65 in both the positive and negative ranges. These two scaled inputs are then summed together before going through the output scaling process and then being output to the actuator. It is clear to see from this definition how the two inputs are "mixed" together before being output to the actuator. This is a simple example of how the mixer system can work, and it is modular enough to support any airframe configuration, as long as the desired allocation of input commands is known.

When building the PX4 firmware for deployment, it is possible to select from a variety of build targets, typically differentiated by the hardware configuration of the target flight controller. If the target is a simulation environment, then a software-in-the-loop (SITL) variant can be built which is essentially identical to the standard variant that runs on all Pixhawk hardware, with specific hardware-based functionalities virtualized so that the firmware can run in a software environment on Linux-based computers. The computer's processor runs all of the Pixhawk's processes and tasks in the same manner that a physical FCU would, and any sensor modules, actuator drivers, and telemetry interfaces present on the physical variants have had modifications made so as to interact with a simulation-based environment rather than via physical interfaces on the FCU. These modifications aside, the overall SITL firmware architecture runs identically to the standard hardware firmware architecture, providing rapid testing capabilities and expediting the Pixhawk platform's development process. The architectural similarity between the hardware-based firmware and the SITL firmware was essential in simulating the Omnicopter's performance so as to provide as seamless a transition as possible when migrating from software simulation to hardware implementation in the future.

While a hardware implementation of the PX4 firmware would communicate directly with the vehicle it controls via hardware connections, the SITL variant routes all standard hardware outputs through MAVLink messages that are then sent to the chosen simulator environment.

The MAVLink protocol is a communications framework for micro air vehicles such as multirotors and small fixed-wing RC vehicles, designed by the creators of Pixhawk specifically for resource-constrained systems and bandwidth-constrained links [25]. Similarly to ROS, MAVLink employs a publish-subscribe system for data, while allowing point-to-point communications for specific sub-protocols. MAVLink was designed for communication between on-board vehicle components and between the vehicle and the ground station, allowing

telemetry and data to be transferred with low overhead costs. Additionally, the MAVROS package is available that allows MAVLink messages to be passed between ROS nodes and MAVLink-enabled hardware and software components.

MAVLink message structures are defined using the XML file format, and can contain multiple data fields of various data types. Like ROS, software components can publish or subscribe to MAVLink topics to exchange data with other software components or the ground station, thereby providing a layer of abstraction between lower-level hardware components and the flight controller or ground station. This benefits developers by allowing them to integrate new software or hardware components onto the micro air vehicle and immediately interface with the rest of the vehicle's components through a generalized communications framework using the various pre-defined and user-defined message types available.

3.2.4 Summary of Simulation and Control Frameworks

The simulation and control frameworks mentioned in this section will be used to ease the implementation of the Omnicopter design. A Pixhawk flight controller, running the PX4 firmware, will be used in the physical vehicle for its plug-and-play characteristics and ease of implementation. The MAVLink protocol will be used in conjunction with ROS to facilitate communication of position and attitude trajectory data between the ground control station and the physical vehicle. Finally, Gazebo will be used as the simulation environment, along with a software-in-the-loop variant of the PX4 firmware, to develop the vehicle simulation.

Chapter 4

Hardware Implementation

This chapter details the work done toward the physical implementation of the Omnicopter vehicle, including component identification, CAD modeling, rapid prototyping, and physical testing and construction.

4.1 Omnicopter Hardware Design

4.1.1 Omnicopter Hardware Components

The physical implementation of the Omnicopter vehicle began by identifying the necessary components to acquire. In their paper, Brescianini and D'Andrea list the names and models of most physical component used on the vehicle. In choosing components for our initial implementation of the vehicle, we sought to maximize the similarity between our vehicle and Brescianini and D'Andrea's vehicle to ensure functionality would not be affected by differences in physical components. Thus, where possible, the components chosen are identical to those used on the original Omnicopter. In the future, as it becomes a necessity and the

operation of the vehicle becomes more familiar to us, it may be possible to change certain physical components to target other performance metrics or enable other capabilities. Table 4.1 lists the components chosen for the initial implementation, whether or not they are effectively the same as those used in the original vehicle, their quantities and their prices.

Table 4.1: Chosen vehicle hardware components.

Part Type:	**Part Name:**	**Matches Original Part?**	**Quantity:**	**Unit Cost:**	**Total Cost:**
Motors	MRM Titan 2208-1100KV	Essentially	8	$14.75	$118.00
ESC	38A EXTREME32 PLUS	Slightly different	8	$13.99	$111.92
Battery	Thunder Power RC TP1800-4SM70	Essentially	1	$21.99	$21.99
Props (left)	Graupner 3D 8×4.5	Yes	8	$1.00	$8.00
Props (right)	Graupner 3D 8x4.5	Yes	8	$1.00	$8.00
Inner Chassis	Carbon Fiber Rods	Essentially	4	$18.98	$75.92
Outer Chassis	Carbon Fiber Rods	Essentially	2	$23.00	$46.00
Diagonal Members	Carbon Fiber Rods	Essentially	2	$20.99	$41.98
Flight Controller	Pixhawk 4 Mini	No	1	$139.00	$139.00
Companion Computer	Raspberry Pi Zero W	Original Part Not Listed	1	$34.50	$34.50
Power Distribution Board	Matek XT60 PDB	Original Part Not Listed	1	$8.49	$8.49
XT60 Splitter	Eachbid Parallel XT60 Connector	Original Part Not Listed	1	$6.66	$6.66
				Overall Cost:	$620.46

For most of the component types listed in Table 4.1, it was possible to choose them such that they were essentially the same as those used in the original Omnicopter. This includes the motors, the battery, the propellers, and the structural components. Of the components listed, the only major differences lie in the flight controller and the ESCs.

Onboard the original Omnicopter, a custom flight controller using an STM32F405x-based microcontroller was constructed to run the control laws and send commands to the motors [2]. For our implementation, we chose to use a commercially available off-the-shelf flight controller to facilitate the integration process. The Pixhawk 4 Mini flight controller is a miniaturized version of the Pixhawk 4 flight controller, which uses the latest Pixhawk FMUv5 open hardware design and runs the PX4 firmware on the NuttX operating system as discussed in Section 3.2.3 [26]. Selecting an off-the-shelf flight controller trades the ability to configure the onboard control system architecture for expedited integration, which was deemed acceptable for our initial implementation of the Omnicopter vehicle. Future iter-

ations may make use of custom flight controller boards to configure the system to achieve different capabilities.

While the original Omnicopter used ESCs using standard analog PWM protocols, we chose ESCs compliant with the DShot ESC to drive the motors. DShot is a digital protocol, unlike typical analog PWM protocols, and provides bi-directional serial communication, meaning telemetry data can be sent from the ESC back to the flight controller [27]. This telemetry includes motor current data and temperature data, but the data of interest is motor angular speed data, which will be fed into the feedback control laws present in Section 3.1.2. The reasoning for selecting these ESCs is primarily the telemetry feature, in addition to the increased signal accuracy and timing provided by the digital protocol. In the original Omnicopter, modifications were made to the ESCs used so that a pulse was sent to the custom-built flight controller on each full commutation of the motors, and by measuring the timing of the pulses the motor angular velocity could be determined. The DShot ESC protocol, which had not been introduced and was thus unavailable to the original authors at the time, provides this functionality in a manner that is already supported by the Pixhawk flight controller and does not require modification of the ESC firmware. Thus, while achieving the same goal, the DShot implementation is a smoother process overall and provides the additional benefits of digital communications between the flight controller and the ESCs.

All of the components listed in Table 4.1 were ordered and acquired as soon as possible, with the exception of the flight controller and seven out of the eight ESCs, which were back-ordered and have not yet been acquired. Thus, any component tests needing the flight controller were run using an on-hand Pixracer flight controller, which runs the slightly older FMUv4 open hardware design [28], but provides the functionality needed to perform hardware tests. These tests included sending actuator commands to the motors to ensure operation, and to test the DShot ESC protocol. By connecting a single motor to the FCU and arming it, it

was determined that this telemetry data is natively sent back into the FCU and published to the `esc_status` uORB topic, where the data can be accessed by the Omnicopter's controller modules running onboard the FCU.

While the original Omnicopter's material choice for its structrual components was specified, the dimensions were not. As such, the carbon fiber rods chosen to comprise the outer chassis were sized using images of the original Omnicopter. Through analyzing these images, we chose 2 mm inner diameter (ID) x 4 mm outer diameter (OD) circular carbon fiber rods for the structural members on the face edges, 3 mm ID x 5 mm OD circular carbon fiber rods for the structural members on the face diagonals, and 8 mm ID x 10 mm OD circular carbon fiber rods for the structural members on the internal cube diagonals. These carbon fiber rods were deemed sufficiently similar when compared to the structural members shown in images of the original Omnicopter, and any discrepancies were deemed negligible due to the carbon fiber rods' ample structural capabilities negating any potential losses in structural rigidity.

4.1.2 Omnicopter CAD Model

Once most of the hardware components were identified, the next step was to produce a detailed three-dimensional CAD model to inform the physical construction and to act as a visual representation of the vehicle during simulation. The model was created using the Autodesk Inventor 2020 software package, by referencing the dimensions, images, and videos provided by Brescianini and D'Andrea in their paper and online [2].

The first portion modeled was the electronics housing in the center of the vehicle. Because this portion was designed and 3D printed by Brescianini and D'Andrea in the original paper, a similar approach was taken in this work. A concept for the central housing was designed

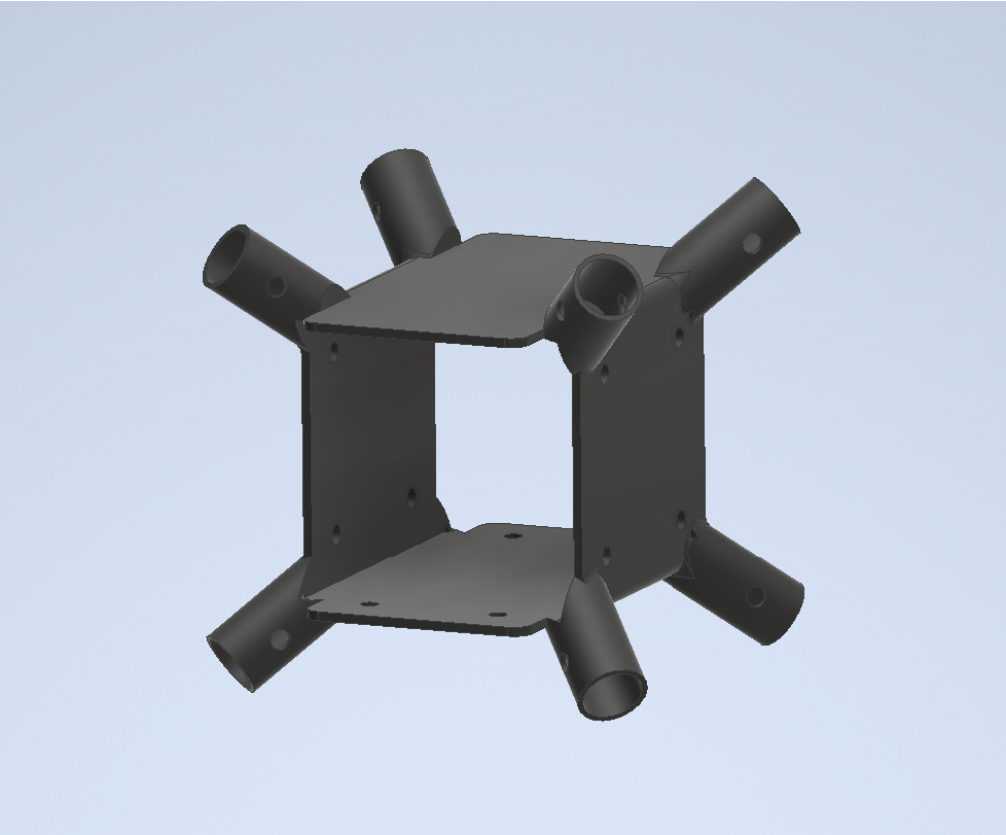


Figure 4.1: Omnicopter central housing design. The flight controller, companion computer, and both power distribution boards will be mounted on the external faces of the housing, with the battery inserted and fastened through the middle opening.

in Inventor to fit a battery of similar dimensions to the one used on the original Omnicopter, with mounting holes placed throughout to accommodate the necessary electronics that the physical vehicle would require. A picture of the 3D-modeled part is shown in 4.1.

The resulting central housing is 50 by 50 by 75 mm, and possesses slots for each of the 10 mm OD carbon fiber rods that create the internal cubic diagonals of the structure. The mounting holes on two of the faces were placed to fit the mounting holes on the two power distribution boards, and on one face to fit the mounting holes of a Raspberry Pi Zero which would act as the companion computer and handle offboard communication via MAVROS. The final face was left smooth since the Pixhawk 4 Mini does not have any external mounting

holes, and instead would be attached by strong adhesives or velcro straps. Additionally, the eight slots for the carbon fiber tubes have holes in them to match with holes drilled into the carbon fiber tubes, allowing the tubes to be fastened to the center housing using a standard M3 bolt.

The next parts to be modeled were the brackets that connect the carbon fiber rods on the corner of the cubic structure. These brackets were modeled to closely resemble those used in the original Omnicopter design, as their configuration in our implementation would be identical to those used in Brescianini and D'Andrea's implementation. Due to the configuration of the structural elements within the Omnicopter's chassis, two variants of the corner bracket were needed, shown in Figure 4.2.

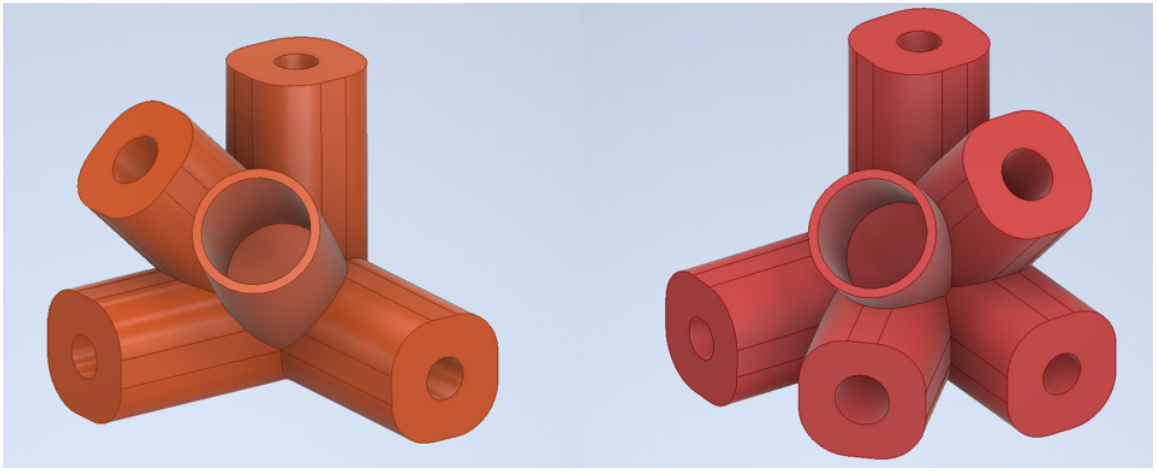


Figure 4.2: Omnicopter corner bracket designs. These brackets hold the external structural members in place.

The two bracket variants differ only in the number of slots they contain for face-diagonal carbon fiber rods; the first has only one face diagonal slot, while the second contains one each on the other two faces. These brackets are each 30 by 30 by 30 mm, with slot dimensions that match the diameters to within 0.13 mm for the outer structural members and 0.250 mm for the inner diagonal structural members in order to produce a snug fit.

The final parts to be designed were the motor clips that fastened the motors to the 10 mm OD carbon fiber rods. Besides the mechanism by which the motor clips attach to the rods, it was difficult to recreate the design used in the original Omnicopter due to a lack of clear images. Thus, the design used in this implementation was created to provide the desired functionality without the use of reference images. A picture of the design is shown in Figure 4.3.

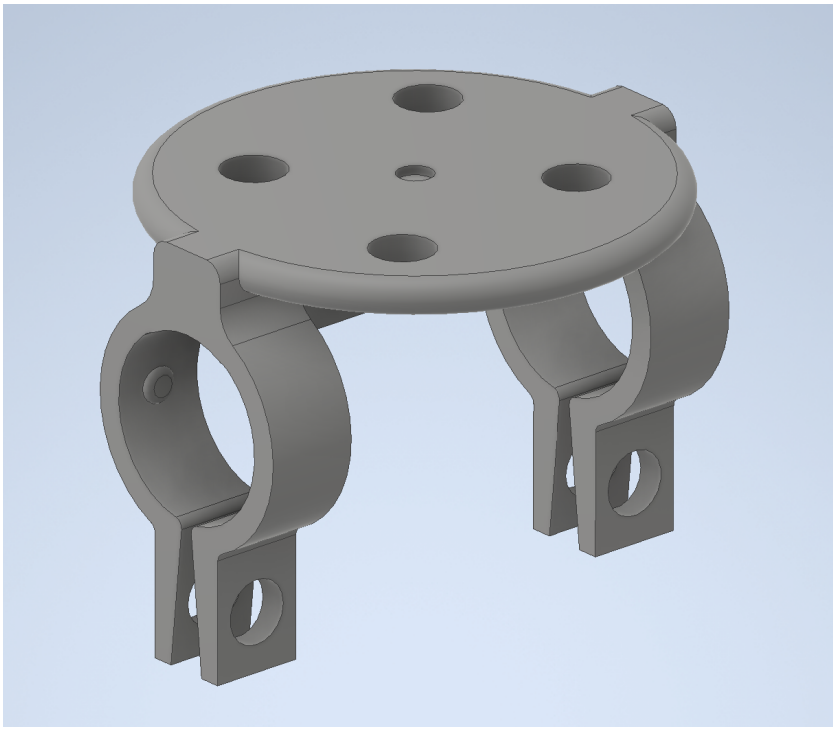


Figure 4.3: Omnicopter motor clip design. The motors fasten into the mounting holes on top of the clip, which is then attached onto the inner diagonal structural members. The protrusions on the interior of the clips fit into holes on the structural members to hold the motors in the correct posiiton and orientation when fastened.

The mounting holes on the surface that interfaces with the motor were placed using dimensions of the motor's mounting holes provided by the motor manufacturer. The motor clips then attach the motor to the 10 mm OD carbon fiber rods by clamping around the exterior and are fastened down by two standard M3 bolts. To prevent the motors from

rotating, small raised protrusions on the interior of each clip fit into holes drilled into the 10 mm OD carbon fiber rods, providing an interference fit when the clips are tightened down. These were drilled using a custom-designed and 3D printed alignment tool to ensure that these holes are aligned such that the motors are secured with their normal vector facing the proper direction.

After designing the necessary parts, 3D representations of the structural components were created in order to construct a full assembly, as shown in Figure 4.4. The assembly shown in this figure is representative of the planned physical vehicle, and the dimensions therein informed the necessary length of each structural elements. The assembly was sized so that the characteristic edge length of the cubic structure matched the value of 0.45 m given in [2]. It is important to note that the rotors are missing from this model, as when implemented into Gazebo the rotors rotate dynamically during the simulation and therefore needed to be separated from the static structural assembly.

4.2 Omnicopter Hardware Assembly

4.2.1 Component Rapid Prototyping

To facilitate the iterative nature of the design process when developing the aforementioned parts, rapid prototyping was used to produce both test and final components. A Creality Ender 3 Pro 3D printer was used to produce the parts, printing in polylactic acid (PLA) filament. PLA filament is affordable, lightweight and sturdy, though has low thermal tolerances when compared to other common 3D printer filaments, but this was not deemed a concern for this particular application. The designs of the 3D printed parts experienced multiple iterations, with tolerances modified until the fit was deemed satisfactory and improvements

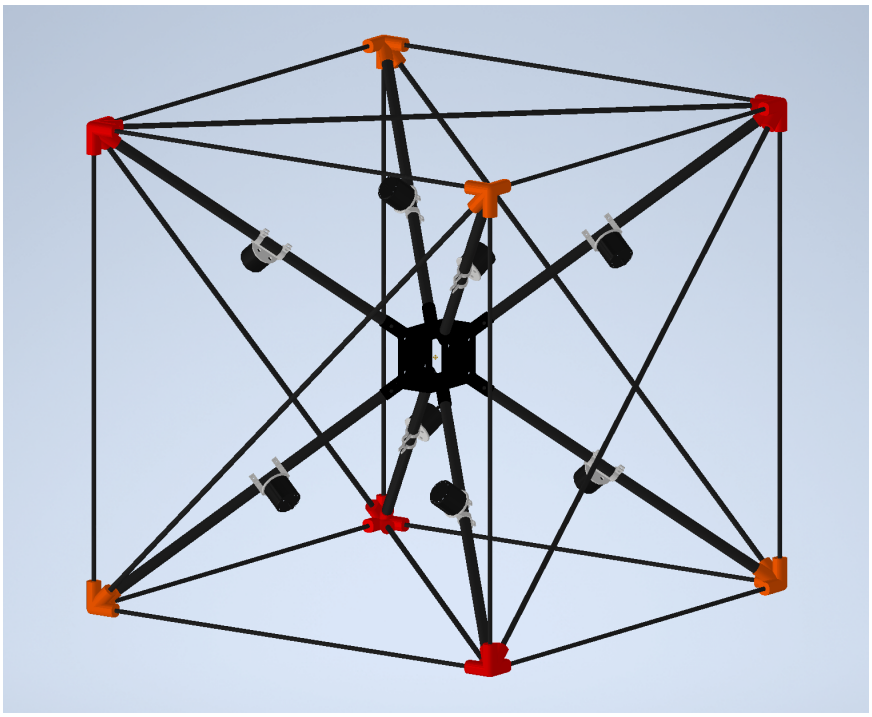


Figure 4.4: Omnicopter 3D CAD assembly. This assembly was created as a scale-accurate representation of the Omnicopter in order to facilitate physical construction and to be used in software simulations.

made where inherent design flaws were identified. The final 3D printed results are shown below in Figure 4.5.

4.2.2 Chassis Assembly and Next Steps

As 3D printed parts were designed and printed, assembly of the main chassis began. The scale-accurate 3D CAD model shown in Figure 4.4 was used in order to determine the necessary lengths for each of the structural components, given in Table 4.2:

Each of the carbon fiber structural members were cut to fit these lengths, and holes were drilled in the correct locations and orientations using a custom designed and 3D printed guide that slotted over the rods. Once the rods were cut to the correct length, test fits

Figure 4.5: 3D printed Omnicopter parts. Top: The two variants of the corner bracket design. Left: The center housing design. Right: The motor clip design.

Table 4.2: Structural component dimensions.

Part Type:	**Length (mm):**
4 mm OD Rod	414.0000
5 mm OD Rod	591.9700
10 mm OD Rod	326.0185

with the 3D printed parts revealed small tolerance issues, and the designs were iterated and re-printed until the fit was satisfactory. After assembling the main structure, the motors and motor clips were fastened using the guide holes cut into the cubic diagonal carbon fiber rods. Small discrepancies in the orientation of the motors were noted due to slight inaccuracies in the machining process which will be corrected before the final physical implementation of the vehicle. The two power distribution boards were fastened to the central housing using the mounting holes set into the design. The Pixhawk 4 Mini flight controller, the ESCs and the companion computer have not yet been integrated, but will be as soon as they are

acquired. The vehicle as it is currently assembled is shown below in Figure 4.6.

Figure 4.6: Current Omnicopter construction progress, showing the fully assembled vehicle chassis, 3D printed components, and motors mounted in the correct orientations.

The next step with regards to physical assembly is to acquire and test the flight controller. This will involve actuating all 8 motors while connected to the FCU, ensuring that DShot motor angular velocity is being received, and testing MAVROS communication using the Raspberry Pi Zero W companion computer. Once these functionalities have been tested, the FCU can be integrated with the vehicle. Then, the ESCs and companion computer will need to be fastened to the central housing, and wiring will need to be organized such that the overall footprint of the central housing is minimized in order to reduce potential aerodynamic interactions with the rotor blades. Finally, a method for securing the battery within the central housing cavity that allows it to be removed easily but remain secure

in flight will need to be developed. Current ideas include padding the interior with high-density foam to hold the battery and then securing it in place with small velcro strips, but the validity of this idea will need to be tested. Once all physical components have been implemented on the vehicle, the vehicle's inertial characteristics such as mass and moment of inertia matrix will need to be measured for use within the controller schemes. Additionally, the rotor's aerodynamic properties will need to be measured through load cell tests and the dynamic behavior of the motors will need to be modeled to insure accurate low-level control. Finally, the controller software will need to be ported to the hardware FCU and tested within a hardware-in-the-loop environment, before finally being tested and tuned onboard the physical vehicle.

Chapter 5

Software Implementation

This section will discuss the steps taken to model the Omnicopter vehicle in the simulation environment, implement the control logic discussed in Section 3.1.2, and simulate the motion of a small satellite in orbit.

5.1 Simulation Overview

A diagram of the simulation architecture used in this thesis is shown in Figure 5.1. The simulation environment is composed of three major components: the ROS/MAVROS nodes, the PX4 SITL firmware, and the Gazebo simulator.

As shown in Figure 5.1, the ROS trajectory node first publishes trajectory data to ROS topics subscribed to by the MAVROS node. MAVROS then takes this data and converts it to relevant MAVLink message types, before sending it to the simulated FCU. The simulated FCU receives the MAVLink messages, decodes them, and publishes their information to the onboard uORB messaging system, where the data is then accessible to any modules

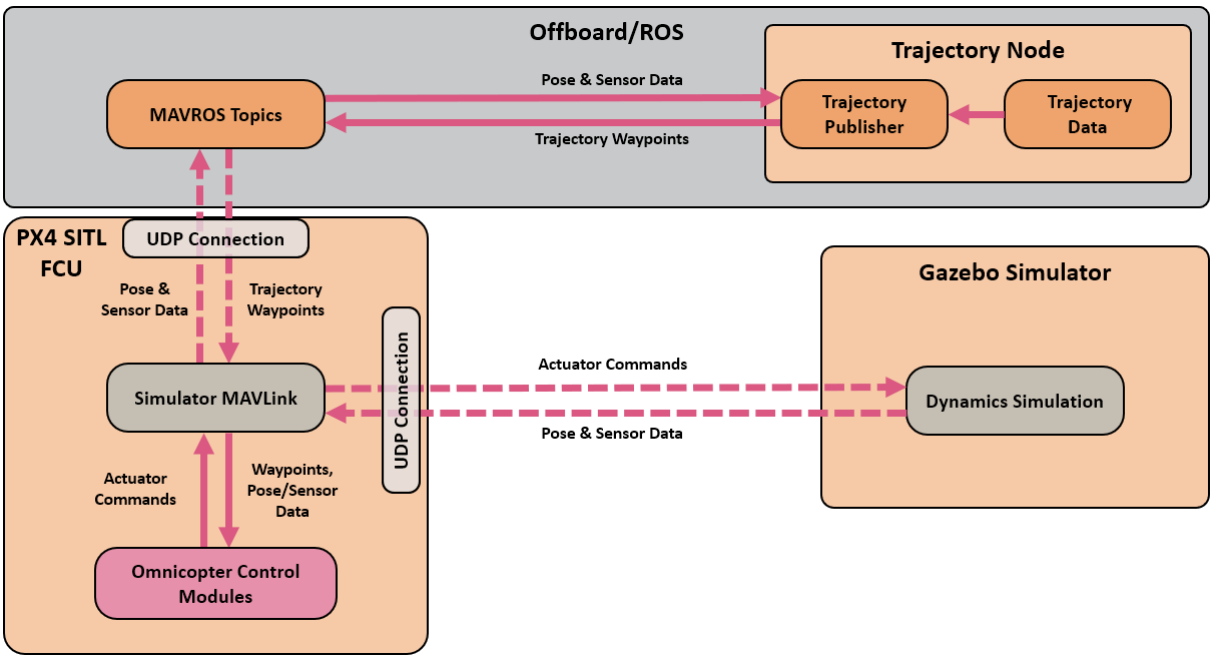


Figure 5.1: Simulation architecture diagram. A high-level diagram of the overall simulation architecture, necessary components, and relevant interfaces.

running onboard the FCU. This data is subscribed to by the controllers and control allocation modules, and actuator outputs are calculated and sent via MAVLink to Gazebo, where the vehicle is simulated dynamically. Information regarding the simulated vehicle's pose, including position and attitude data, is then returned to the FCU where it is used for feedback control and to the trajectory node where it is used to determine whether or not the vehicle has reached a waypoint. This general system architecture encompasses many lower-level modules and components that enable the simulation to run; these components which are relevant to the content of this thesis will be discussed in detail in the following sections, beginning with the work done toward integrating the Omnicopter vehicle into the Gazebo simulation environment.

5.2 Gazebo Simulation Environment

5.2.1 Omnicopter Gazebo Model

Defining the Omnicopter model in Gazebo was the first step taken in developing the simulation. As discussed in Section 3.2.1, Gazebo uses the SDF file format to define models. SDF is an XML-like format that defines physical model parameters within text block structures. SDF files are typically divided into the definitions of various links and joints, followed by the declaration of any included plugins. In most SDF files, the first definition block is the "base link", or the base-level component in the model that each other component is derived from. For the Omnicopter, the base link represents the Omnicopter's base structure, independent of the rotors or other moving components. The SDF code that creates and defines the physical parameters for the base link is shown on the next page.

```
<model name='omni'>
  <pose>0 0 0.225 0 0 0</pose>
  <link name='base_link'>
    <velocity_decay>
      <linear>0.0</linear>
      <angular>0.0</angular>
    </velocity_decay>
    <inertial>
      <pose>0 0 0 0 0 0</pose>
      <mass>0.892</mass>
      <inertia>
        <ixx>0.0116</ixx>
        <ixy>0</ixy>
        <ixz>0</ixz>
        <iyy>0.0113</iyy>
        <iyz>0</iyz>
        <izz>0.0113</izz>
      </inertia>
    </inertial>
    <collision name='base_link_collision'>
      <pose>0 0 0 0 0 0</pose>
      <geometry>
        <box>
          <size>0.45 0.45 0.45</size>
        </box>
      </geometry>
      <surface>
        <contact>
          <ode>
            <max_vel>100.0</max_vel>
            <min_depth>0.001</min_depth>
          </ode>
        </contact>
        <friction>
          <ode>
            <mu>1.0</mu>
            <mu2>1.0</mu2>
          </ode>
        </friction>
      </surface>
    </collision>
    <visual name='base_link_visual'>
      <geometry>
        <mesh>
          <scale> 1 1 1 </scale>
          <uri>model://omni/meshes/omni.dae</uri>
        </mesh>
      </geometry>
      <material>
        <script>
          <name>Gazebo/DarkGrey</name>
        </script>
      </material>
    </visual>
  </link>
```

Source Code 2: Omnicopter SDF File Excerpt

The first keyword, `<model name = 'omni'>`, denotes the beginning of the file for the mdoel named omni. Below this, `<pose>0 0 0.225 0 0 0</pose>` denotes the base link's positional and angular offsets relative to it's coordinate system. The first three numbers represent x, y, and z offsets in meters, and the last three represent angular rotations about the three axes in radians. In this case, the entire Omnicopter model is offset 0.225 meters in the z-direction when inserted into the world to account for the dimensions of the model. The next block of code, beginning with `<link name='base_link'>`, is where the physical parameters of the base link are defined. The `<inertial>` block defines the translational and rotational inertia parameters, such as mass and the inertia matrix. After this, the `<collision>` block deals with the physical collision properties of the model, creating the bounding box that interacts

physically with the simulated world. Finally, the `<visual name='base_link_visual'` block loads the 3D CAD model of the Omnicopter and applies it to the base link to give the model its visual representation in the world.

At the end of the above block of code, the Omnicopter's chassis will be defined and implemented in the simulation environment. When spawned in, however, the vehicle will not respond to commands or possess any rotors that can be actuated. To implement the motors, both rotor links and plugins are necessary. The rotor links are the physical and visual elements of the rotors within the simulation, and the plugins will drive the rotation of the rotors and produce the corresponding thrusts and torques that will be applied to the Omnicopter model. Each rotor link's definition within the SDF file is structurally similar to the base link code shown above, with the mass and inertial attribute values corresponding to the physical rotors and each `<pose>` line containing the corresponding position and orientation values from Equations 3.3 and 3.4. The rotor links each have an additional element block describing the behavior of the joint. An example of this additional block is shown on the next page for the first rotor.

```
<joint name='rotor_1_joint' type='revolute'>
  <child>rotor_1</child>
  <parent>base_link</parent>
  <axis>
    <xyz>-0.788675 0.211325 0.57735</xyz>
    <limit>
      <lower>-1e+16</lower>
      <upper>1e+16</upper>
    </limit>
    <dynamics>
      <damping>0.004</damping>
    </dynamics>
    <use_parent_model_frame>1</use_parent_model_frame>
  </axis>
  <physics>
    <ode>
      <implicit_spring_damper>1</implicit_spring_damper>
    </ode>
  </physics>
</joint>
```

Source Code 3: Rotor SDF Joint Block Example

The first line defines the joint name and type. In this case, the joint is a "revolute" joint, meaning that the only movement allowed is rotational motion about the central axis of the joint. The joint axis defined in the line beginning with `<xyz>` to be inline with each rotor's normal axis defined in Equation 3.4. The `<limit>` parameter sets the minimum and maximum rotational speeds for the joint, which in this case are all set to numbers far beyond the expected maximum and minimum speeds so as not to limit the performance of the vehicle. The other parameters specified are standard parameters used in joint definitions. After including all eight rotor link definitions, the vehicle model is ready to be populated with plugins that allow it to be controlled by the Pixhawk flight controller. An image of the Omnicopter Gazebo model is shown in Figure 5.2.

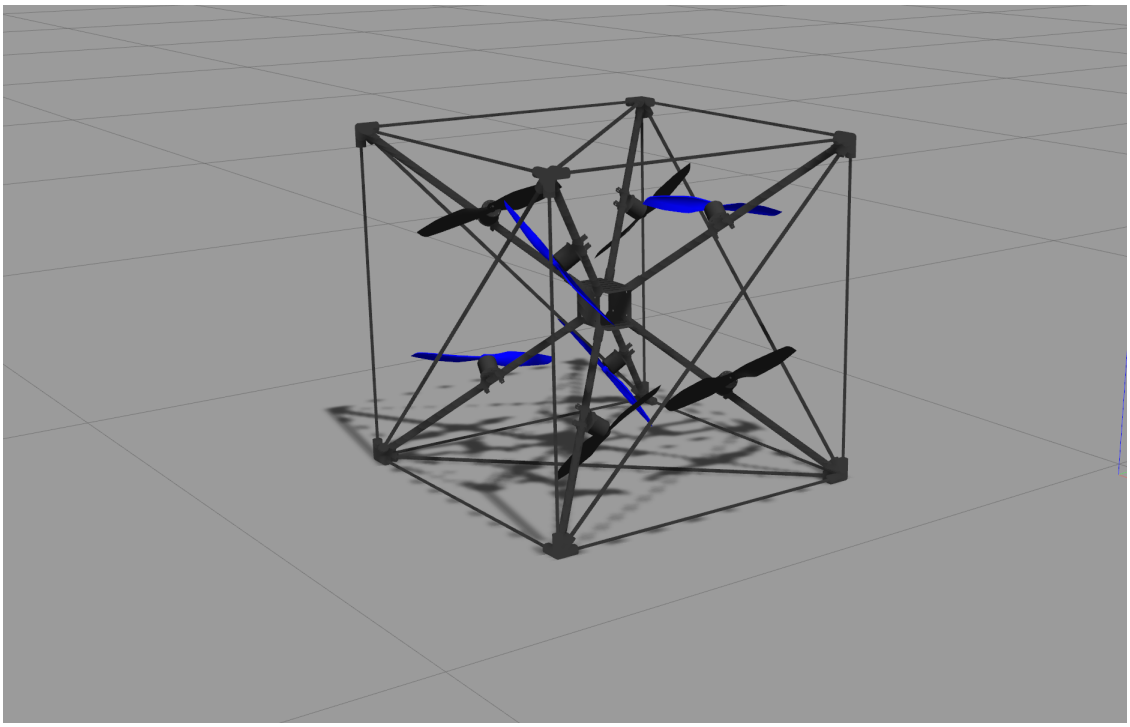


Figure 5.2: Gazebo Omnicopter model, as defined by the omni.sdf file and visualized using the CAD model discussed in Section 4.1.2.

5.2.2 Gazebo Motor Models

To actuate the rotors present in the vehicle model, motor plugins are used. These plugins are included as a part of the `sitl_gazebo` software package released by the developers of the Pixhawk flight controller. This package provides Gazebo plugins that allow the FCU to interact with the vehicle being simulated. This includes modeling sensor hardware such as inertial measurement units (IMUs), simulating GPS measurements, and providing interfaces for communicating via MAVLink. Included in this package is the `gazebo_motor_model` plugin, which provides an interface between the commanded motor outputs provided by the FCU and the rotor joints within the Gazebo model.

It is important to note that apart from the motor plugin, Gazebo has minimal built-in aerodynamics modeling. The aerodynamic forces and moments produced by the rotors are

calculated by the `gazebo_motor_model` plugin as functions of the rotor joints' angular velocities using momentum-blade element theory and are then applied directly to the rotor joints [21]. This adequately simulates the forces and torques produced by the rotors of a real multi-rotor vehicle.

The force produced by the ith rotor, $f_{rotor,i}$, is calculated using the following expression:

$$f_{rotor,i} = c_{f,i}\text{sgn}(\Omega_i)\Omega_i^2 \tag{5.1}$$

where Ω_i is the angular velocity of the ith rotor, and $c_{f,i}$ is the force coefficient for the ith rotor, which has units of $\frac{N}{(rad^2/s^2)}$. This force is applied normal to the rotor disk plane.

As each of the rotors spin, a drag torque is produced by the blades moving through the air which opposes the rotor's spin direction. This aerodynamic drag torque is modeled by the following expression:

$$t_{rotor,i} = -c_{t,i}\text{sgn}(\Omega_i)\Omega_i^2 \tag{5.2}$$

where $c_{t,i}$ is the rotor torque coefficient, with units of $\frac{N-m}{(rad^2/s^2)}$. If κ_i is defined as the thrust-to-drag ratio for the ith rotor, $\kappa_i = \frac{c_{f,i}}{c_{t,i}}$, then the previous expression for $f_{rotor,i}$ can be used to re-write the ith rotor's drag torque as

$$t_{rotor,i} = -f_{rotor,i}\kappa_i \tag{5.3}$$

The implementation of the above two equations into the `gazebo_motor_model.cpp` source file is shown below, occurring on lines 196 and 235, respectively.

```
double force = real_motor_velocity * real_motor_velocity * motor_constant_;

ignition::math::Vector3d drag_torque(0, 0,
                        -turning_direction_ * force * moment_constant_);
```

The `turning_direction_` variable is defined to be $+1$ for counter-clockwise (CCW) spinning motors and -1 for clockwise (CW) spinning motors, corresponding to the direction of spin that produces positive thrust. It is important to note that the force and moment calculations here only account for motors that spin in a single direction; that is, they do not take into account the current sign of the rotor's angular velocity, but assume that a CCW motor will always spin CCW. The Omnicopter model makes use of symmetric or "3D" propellers, which can produce thrust profiles that are equal in magnitude but opposite in direction when spinning either CCW or CW. Thus, to allow for these bidirectional thrusts (and resulting torques) to be generated, the plugin's source code file was copied and renamed `gazebo_motor_model_edits.cpp`, with the edits shown below inserted in place of the above code.

```
double force = rot_sign_ * turning_direction_ * real_motor_velocity
                                * real_motor_velocity * motor_constant_;

ignition::math::Vector3d drag_torque(0, 0,
                                -rot_sign_ * force * moment_constant_);
```

The first change made was to the force calculation, with the addition of the `rot_sign_` term. In `C++`, the sign of the rotor's angular velocity can be determined using the logical expression in Equation 5.4.

$$\text{sgn}(\Omega_i) = (\Omega_i > 0) - (0 > \Omega_i) \tag{5.4}$$

When evaluating this expression, if $\Omega_i > 0$, then the first parenthetical expression evaluates to $+1$ and the second evaluates to 0, thereby returning $\text{sgn}(\Omega_i) = 1$, which denotes a CCW rotation. Likewise, if $\Omega_i < 0$, then the first parenthetical expression evaluates to 0 and the second evaluates to $+1$, which is then negated, returning $\text{sgn}(\Omega_i) = -1$, which denotes a CW rotation. This value is then multiplied by the turning direction in the original expression for force to correct the sign on the resulting force depending on the direction of the motor's rotation.

The torque expression is modified as well, replacing the variable `turning_direction_` with the sign of the rotor's rotation, `rot_sign_`, to ensure the torque value is opposite the direction of rotation, not just the direction corresponding to positive thrust. The angular velocities of each motor joint are increased and decreased as functions of the throttle values output by the simulated FCU, and the resulting angular velocites are passed through the aforementioned calculations to determine the resulting forces and torques caused by each rotor. Once determined, these forces and torques are applied on each motor's respective joint in the Gazebo enviornment, thereby simulating the behavior of a real rotor.

5.3 PX4 Simulation Components

5.3.1 PX4 SITL

On its own, the Gazebo Omnicopter model possesses no flight control capabilities; thus, to command the vehicle, the SITL variant of the PX4 firmware was used. The first step in integrating the Omnicopter vehicle with the PX4 firmware was to define a custom airframe within the PX4 environment, which means defining a mixer file.

Because the Omnicopter has its own control allocation scheme that is dependent on the vehicle's orientation, using the standard mixer system would not suffice, as it assumes the control allocation scheme for a specific actuator configuration does not change over time. Thus, the Omnicopter's mixer file was designed to act as a "passthrough" mixer. Whereas standard multi-rotor vehicle mixers take normalized roll, pitch, yaw, and thrust inputs generated by controller modules that are then allocated to relevant actuators, the Omnicopter's controller modules output already allocated actuator conrol inputs, which means its mixer needs to simply route the commands to the correct actuator without modifying their values. In the sense of Equation 3.32, the Omnicopter's mixer can be represented by the following matrix equation:

$$\begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ u_8 \end{bmatrix} = \boldsymbol{I}_{8x8} \begin{bmatrix} u_{1,cmd} \\ u_{2,cmd} \\ \vdots \\ u_{8,cmd} \end{bmatrix} \tag{5.5}$$

where the mixer matrix is analogous to the 8-by-8 identity matrix, allowing the individual rotor commands to be sent directly through to their respective actuators without having to bypass the mixer system. The implementation of the Omnicopter's mixer can be seen below.

```
M: 1
S: 0 0  10000  10000      0 -10000  10000

M: 1
S: 0 1  10000  10000      0 -10000  10000

M: 1
S: 0 2  10000  10000      0 -10000  10000

M: 1
S: 0 3  10000  10000      0 -10000  10000

M: 1
S: 0 4  10000  10000      0 -10000  10000

M: 1
S: 0 5  10000  10000      0 -10000  10000

M: 1
S: 0 6  10000  10000      0 -10000  10000

M: 1
S: 0 7  10000  10000      0 -10000  10000
```

Source Code 4: Omnicopter Mixer Code

Each actuator block takes as input a single unique control channel from Control Group #0. The scalings are defined such that no modifications are made to the input values, and the values are then output to their corresponding actuators, thus "passing-through" the values output by the controller modules directly to the actuators.

5.3.2 Controller Modules

To implement the controllers defined in Section 3.1.2, custom software applications were developed as modules within the PX4 firmware. A module was written for each of the major components of the control architecture shown in Figure 3.3, with some modifications.

The first modification is the inclusion of the position and attitude controllers onboard the FCU, rather than offboard. This change was made with future developments in mind, particularly with regards to smallsat testing and development, as running the control algorithms onboard is how smallsats operate and more closely reflects the operating environment being simulated. It is important to note that the performance of the control algorithms can suffer from latency as a result of sending position and attitude measurements from the motion capture system to the vehicle, but as the physical implementation has not yet been completed this will be investigated at a later date.

The next modification is the combination of the outer attitude controller and the inner angular velocity controller into a single module. This modification was made because it did not seem beneficial to keep them separate after deciding to move the attitude controller onboard. While both controllers are still in operation individually, they have been combined to run within a single task in the PX4 firmware, meaning they are executed simultaneously at the same rate.

The final modification, or rather lack of implementation, is the exclusion of the low-level motor dynamics controller (Equation 3.29) and any motor angular velocity feedback from the implemented control architecture. Extracting the rotor angular velocities from Gazebo and passing them through to the FCU requires further exploration into Gazebo's plugin capabilities and the development of custom MAVROS plugins and MAVLink messages. While these developments can be made with further work, time restrictions inhibited their inclu-

sion into the implementation of this thesis, and as such control algorithms dependent on these values were modified to represent an older variant of the Omnicopter that had not yet considered low-level motor control [29]. This was deemed acceptable for the scope of this thesis as a means to prove the Omnicopter's capabilities of simulating small satellite dynamics, and future work will be dedicated toward implementing any excluded details.

A diagram of the implemented controller architecture is shown below in Figure 5.3.

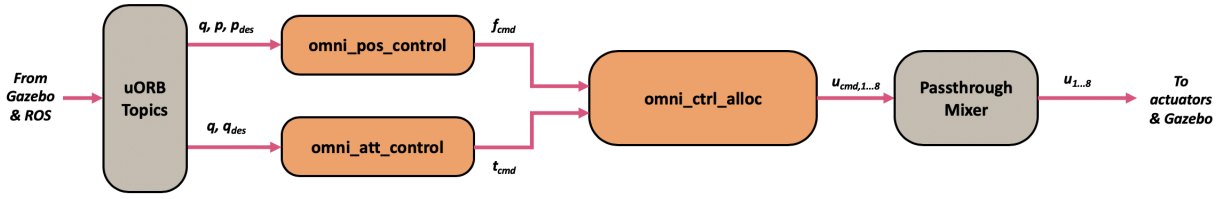


Figure 5.3: Control modules diagram. A block diagram view of the information flow through the Omnicopter control module applications (in orange) and their interactions with native PX4 modules (in grey).

Position Control Module

The `omni_pos_control` module implements the position control logic detailed in Equation 3.6. A block diagram of the module's internal method calls is shown in Figure 5.4.

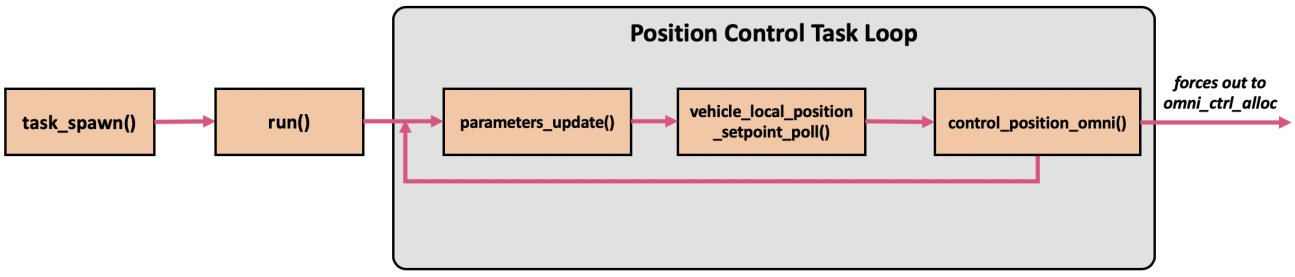


Figure 5.4: Position control module diagram. A block diagram view of the internal method calls within the position control module.

When instantiated, the module's loop is entered through the `task_spawn()` method. This method is a standard part of most PX4 modules and handles the creation of a task within the task scheduler, which ensures that the `run()` method executes in its own NuttX task on a hardware implementation and on its own thread in an SITL implementaiton. The `run()` method begins by creating the uORB topic subscribers and assigning a loop rate of 50 Hz. Next, a loop is entered that exits only when the scheduled task stops running. The first method called is the `parameters_update()` method, which checks to see if any internal PX4 parameters that are used by the `omni_pos_control` module have changed since the last iteration, and updates them if so. Before beginning the controls portion of the loops, the `vehicle_local_position_groundtruth` topic is polled for changes. If the vehicle's position is unchanged, then the loop continues without running the controller, as this would be a waste of processor cycles. If the position has changed, the `vehicle_local_position_setpoint_poll()` method is called. This polls the `vehicle_local_position_setpoint` uORB topic and checks for updates to the currently published position setpoint. If the position setpoint changes, the method copies it to a local variable to be used in the control loop. If the setpoint is unchanged, then the method simply exits without performing any action and the loop continues. This method is used to ensure that processor loop cycles are not wasted updating the position setpoint if there has been no change, as the setpoint only updates once the vehicle has reached the previous setpoint. After polling the position setpoint, the vehicle's position and attitude are updated from the `vehicle_local_position_groundtruth` and `vehicle_attitude_groundtruth` uORB topics. These values are updated with each loop iteration, as accurate and up-to-date position and attitude values are necessary for the position control loop. Finally, once all of these values are updated, the `control_position_omni()` method is called.

The `control_position_omni()` method is where all control logic occurs. This method takes

as input the current position, the current orientation, and the desired position setpoint, and outputs forces in the body frame that the Omnicopter must produce in order to track the desired position setpoint. The calculations occurring inside this method are identical to those expressed in Equation 3.6. Anti-windup checks were implemented before the integral terms are updated by allowing integral updates only when the integrators are not saturated. Once the required body forces are calculated, the method publishes them to a custom uORB topic `omni_body_forces`, which is then subscribed to by the `omni_ctrl_alloc` module.

Attitude Control Module

The `omni_att_control` module implements the attitude and angular velocity control logic detailed in Equations 3.12 and 3.15. A block diagram of the module's internal methods is shown in Figure 5.5.

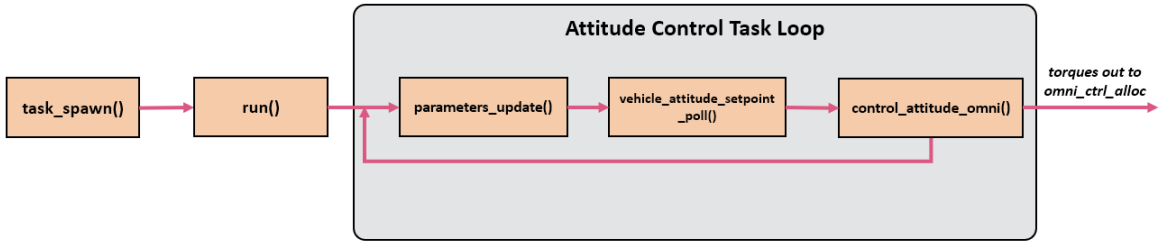


Figure 5.5: Attitude control module diagram. A block diagram view of the internal method calls within the attitude control module.

The structure of the attitude control module is nearly identical to that of the position control module. It also enters the control loop through the `task_spawn()` method, which creates a scheduled task to ensure the module is run on time on its own thread within the operating system. This scheduled task calls the `run()` command, which creates the necessary uORB topic schedulers and sets the loop rate at 250 Hz. While the offboard attitude controller runs at 50 Hz in the original Omnicopter implementation, the onboard angular velocity controller ran at 250 Hz, and since the two controllers are combined onboard

in our implementation it was chosen that this module would run at the faster 250 Hz so as to provide the necessary bandwidth for the lower-level angular velocity controller. The method then enters into a loop that exits only when the scheduled task exits. Next, the `vehicle_attitude_groundtruth` topic is polled to see if the vehicle's attitude has changed. If it has not, then the controller is not run. As with the position controller module, the next method called is the `parameters_update()` method which checks for changes in parameters stored in the FCU's memory. Next, the `vehicle_attitude_setpoint_poll()` is called to check if the attitude setpoint has been changed since the last loop iteration. If so, the new setpoint is copied, but if not, the loop continues. Next, the vehicle's current angular velocity and attitude are polled from the `vehicle_angular_velocity_groundtruth` and `vehicle_attitude_groundtruth` topics. Finally, the `control_attitude_omni()` method is called.

The `control_attitude_omni()` method, as with the `omni_pos_control` module, is where most of the controller implementation takes place. The method implements the calculations detailed in Equation 3.12 first, then feeds the desired angular velocity outputs into the calculations detailed by Equation 3.15. It is important to note that the quaternions representing the vehicle's attitude and the desired attitude do not share the same reference frame. The vehicle's orientation retrieved from the `vehicle_attitude_groundtruth` uORB topic is measured in the north-east-down (NED) vehicle body frame by the simulated IMU included in the Gazebo model definition, whereas the desired orientation quaternion retrieved from the `vehicle_attitude_setpoint` uORB topic is defined in the right-handed east-north-up (ENU) frame before being transformed by MAVROS into the NED frame, which is the FCU's standard frame of reference. Thus, to recover each quaternion in the right-handed ENU vehicle body frame as is assumed by the controllers, the setpoint quaternions need to be transformed. The change in axes orientation when transforming between NED and ENU

is shown in Figure 5.6.

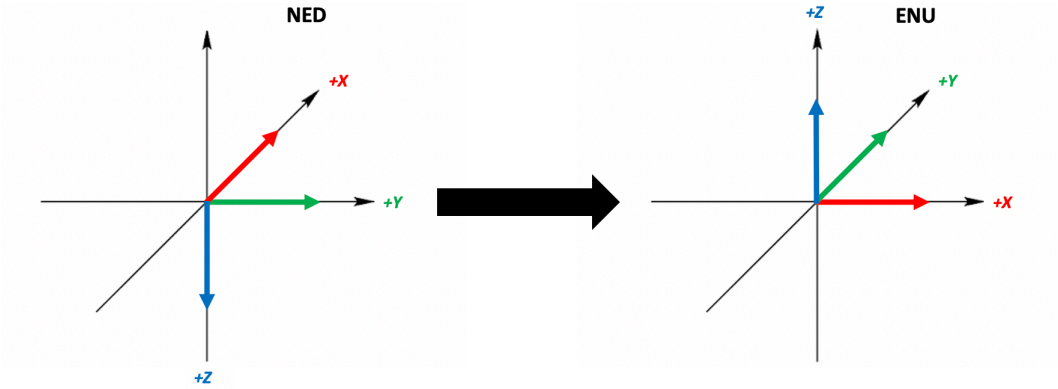


Figure 5.6: NED to ENU transformation. To transform between the two frames, the X and Y axes are swapped, and the Z axis is negated.

In practice, this transformation is accomplished by swapping the X and Y values and negating the Z values for each quaternion. This is sufficient for the vehicle attitude quaternion, but is not enough for the desired quaternion setpoint, which is not defined in the body frame. Thus, a similarity transformation is performed to preserve the direction of the desired quaternion but to transform it into the vehicle body frame:

$$\boldsymbol{q}^b_{\boldsymbol{des}} = (\boldsymbol{q}^b)^{-1} \cdot \boldsymbol{q}^w_{\boldsymbol{des}} \cdot \boldsymbol{q}^b \tag{5.6}$$

where the w supersript denotes a quaternion in the world frame and the b superscript a quaternion in the body frame. Once this transformation is complete, the above method of swapping the X and Y values and negating the Z values of the quaternion can be applied. This ensures that the error quaternion that encodes the rotation between the current orientation and the desired orientation, needed by Equation 3.12, is valid with respect to the vehicle's current orientation. This value is then fed into the proportional and integral terms

of the controller as defined in Equation 3.12 to output the necessary body frame angular velocities. Anti-windup checks similar to those used in the `omni_pos_control` module were also included in this module.

When testing the results of the angular velocity controller, it was noted that the proportional and integral terms were not sufficient to track the desired orientations without oscillation within the Gazebo simulation. While this could be attributed to the lack of low-level motor control in this implementation, this was quickly corrected by including a derivative term in the angular velocity controller, with a derivative gain $k_{att,d} = 1.5$ applied. This term was found to be sufficient in damping the observed oscillations, and its necessity will be re-evaluated once the low-level motor controller is implemented in the future.

After the desired angular velocities are calculated, these values are immediately passed into the angular velocity controller logic detailed in Equation 3.15. Since low-level motor control is not being implemented at this time, the last term of Equation 3.15 that utilizes the individual rotor angular velocities was left out, which is similar to the controller used in the older implementation of the Omnicopter shown in [29]:

$$\boldsymbol{t_{cmd}} = \frac{1}{\tau_\omega}\boldsymbol{J}(\boldsymbol{\omega_{cmd}} - \boldsymbol{\omega}) + \boldsymbol{\omega} \times (\boldsymbol{J}\boldsymbol{\omega}) \tag{5.7}$$

Once the torque values are calculated, they are constrained between -0.3 and 0.3 N-m to prevent large torques from causing coupling effects. These torques are then published to the `omni_body_torques` uORB topic, which are then subscribed to by the `omni_ctrl_alloc` module.

Control Allocation Module

The final module, `omni_ctrl_alloc`, implements the control allocation scheme detailed in Equations 3.23 through 3.27. A block diagram of the module's internal methods is shown in Figure 5.7.

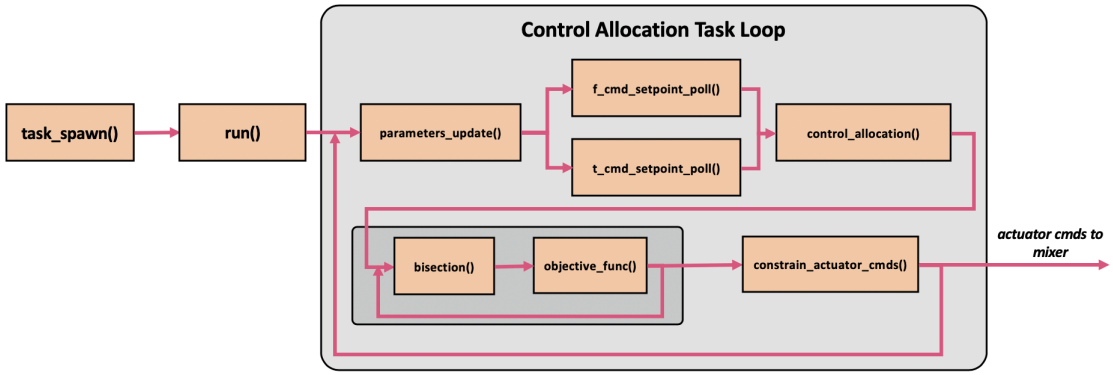


Figure 5.7: Control allocation module diagram. A block diagram view of the internal method calls within the conrol allocation module.

As before, this module starts by entering through the `task_spawn()` method into the `run()` method. The `run()` method begins by creating the necessary uORB subscribers and setting the loop rate to 250 Hz. While the original Omnicopter's implementation set this loop's rate to 1000 Hz, the manner in which most modules are implemented within the PX4 firmware limits the maximum loop rate to 250 Hz. This is because module loops are designed to only execute when a specified uORB topic is updated so as not to waste processor cycles, and the fastest uORB topic updates at 250 Hz. Thus, until a way around this is discovered, the loop rate was chosen to be the maximum loop rate of 250 Hz. Next, `parameters_update()` is called as with the other modules. Because the `omni_ctrl_alloc` module depends on both the commanded forces from the `omni_pos_control` module and the commanded torques from the `omni_att_control` module, the loop controller is allowed to run if either of the `omni_body_forces` and `omni_body_torques` uORB topics have been

updated. Once the loop is entered, the aforementioned uORB topics are polled using the `f_cmd_setpoint_poll()` and `t_cmd_setpoint_poll()` methods to check if the setpoints have changed. After these have been checked, the `control_allocation()` method is called.

The `control_allocation()` method implements the calculations for the the control allocation scheme detailed in Equations 3.23 through 3.27. Due to the lack of low-level motor control, the optimization scheme that minimizes motor reversals in the short term and power usage in the long term is not run during the simulation, though the methods are implemented and will be discussed. First, the suboptimal rotor forces $\hat{\boldsymbol{f}}_{\boldsymbol{cmd}}$ are calculated by evaluating $\hat{\boldsymbol{f}}_{\boldsymbol{cmd}} = \boldsymbol{B}^{\dagger}\boldsymbol{\nu}_{\boldsymbol{cmd}}$. Next, the biases ϕ_1 and ϕ_2 are calculated by minimizing the objective function in Equation 3.27. This is achieved using the bisection method on the derivative of the objective function with respect to each bias in turn, implemented as the `bisecion()` and `objective_func()` methods respectively. The `bisection()` method is called twice in each loop, once for ϕ_1 and once for ϕ_2, and the optimal biases would be used in Equation 3.23 if lower-level motor control were being implemented. Since it is not currently, the `bisection()` and `objective_func` methods are not utilized, and the individually allocated rotor commands are given by $\hat{\boldsymbol{f}}_{\boldsymbol{cmd}} = \boldsymbol{B}^{\dagger}\boldsymbol{\nu}_{\boldsymbol{cmd}}$. Again, since lower-level motor control is not being implemented, rather than converting these rotor force commands to rotor angular velocity commands and sending them to the motor angular velocity controller, the force commands are normalized by the maximum possible rotor thrust and then constrained to lie within the range -1 to $+1$ by the `constrain_actuator_cmds()` method. These final actuator commands are then published to the `actuator_controls_0` uORB topic, which corresponds to the Flight Group #0 control group utilized by the Omnicopter mixer detailed in Section 5.3.1.

5.4 Trajectory Generation and Publishing

To supply the necessary position and attitude setpoints, codes needed to be developed that would generate the trajectories and send them to the FCU. The current implementation reads trajectory setpoints in from a text file and publishes them to the FCU, where each line in the text file represents a setpoint. Trajectory file generation was handled in `MATLAB`, while the trajectory publishing portion was implemented via ROS/MAVROS using Python.

5.4.1 Trajectory Generation

A set of `MATLAB` scripts and functions were written to generate the text files needed for the trajectory publisher. These scripts are currently capable of generating three different classes of position trajectories and three different classes of attitude trajectories, based on user specifications. Additionally, a preview of the generated position and attitude trajectory is displayed for validation.

The main .m file, `OmniOrbitGeneration.m`, handles most of the orbit generation with the assistance of a few helper functions, `eul2quat.m`, `quat2eul.m`, and . The user specifies the name of the position and attitude trajectory types they would like to follow. For positional, the options are "orbit" for a standard circular or two-body trajectory, "rect" for a rectangular trajectory, and "up" for a simple rise-and-hover command. For attitude, the options are "fixed" for an attitude trajectory that maintains zero pitch/roll/yaw throughout, "orbitcenter" for a trajectory that causes the Omnicopter's positive X face to point toward the center or focus of the positional trajectory, and "custom" for a user-specified range of roll, pitch, and yaw angles. Once the type of trajectories are set, specifics about the trajectories can be chosen such as the classical orbital elements for two-body orbits, side lengths

and heights for rectangular trajectories, roll, pitch, and yaw angles for fixed and custom trajectories, etcetera. Additionally, depending on the type of orbit, the code will generate acceleration values at each setpoint that satisfy the kinematics of the desired orbit, which act as the feed-forward term $\ddot{\boldsymbol{p}}_{\boldsymbol{des}}$ in Equation 3.6. For example, in a trajectory representing a two-body orbit, the two-body equation of motion $\ddot{\vec{r}} = -\frac{\mu}{r^3}\vec{r}$ is used, with $\mu = 2.9203e+03$ experimentally determined to provide satisfactory results. Once all trajectory specifications have been made, a preview animation of the orbit is shown to verify the position and attitude trajectories have been properly generated, as shown in Figure 5.8.

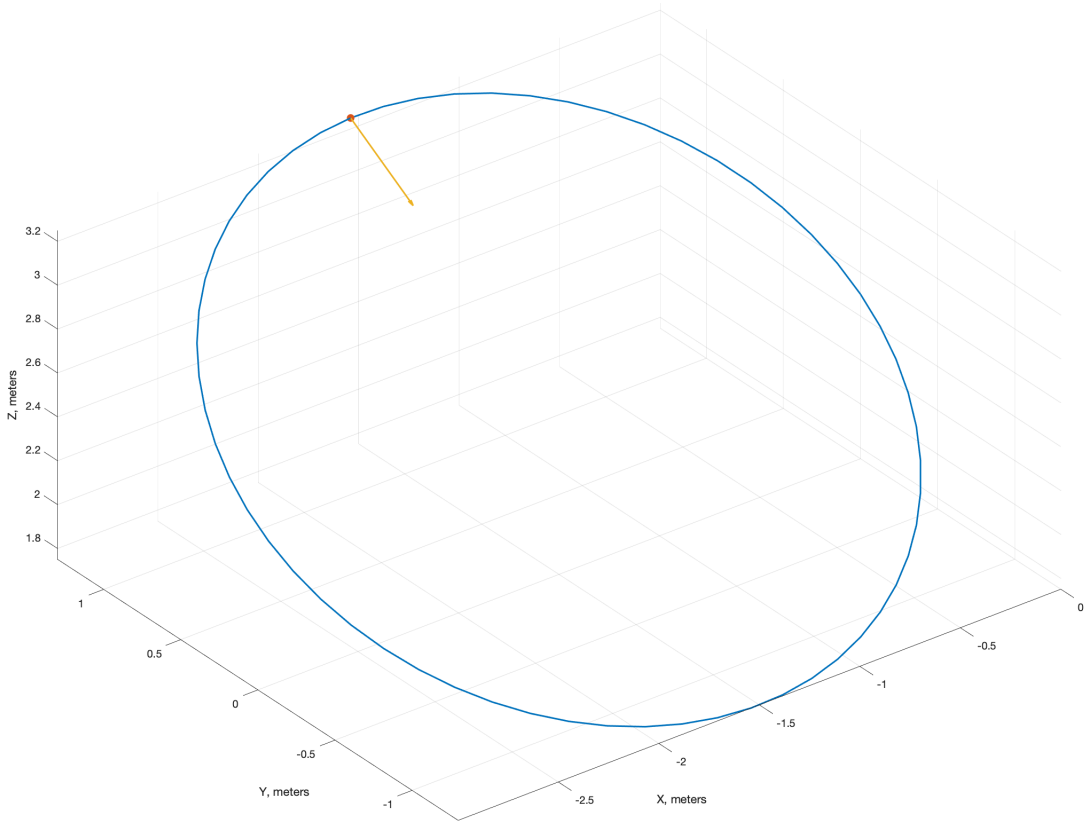


Figure 5.8: Trajectory generation code output. This figure shows an example trajectory generated using the "orbit" positional trajectory type, with 30 degrees of inclination and a semi-major axis of 1.5 meters, and the "orbitcenter" attitude trajectory type.

Once the desired trajectory is verified, the position setpoints and attitude setpoints (in the

form of quaternions) are written to a .txt file, which can then be copied over into the working directory of the trajectory publisher for use in the simulation.

5.4.2 Trajectory Publishing

The trajectory publishing module is a ROS node written in Python that reads in the aforementioned trajectory files and publishes the position, attitude, and desired acceleration setpoints to the FCU via MAVROS. The code begins by reading in the .txt trajectory file specified by the user and formatting the position, velocity, and acceleration setpoints (if present) into a Python list. The list of trajectory data is then fed into the `traj_pub()` method which handles the majority of the trajectory publishing. Upon entering this method, the initial position setpoint is formatted into a MAVROS `PositionTarget` message and the attitude setpoint into standard ROS `PoseStamped` method using the `getSetpoint()` method, which takes the list of trajectory data and the current line index as input. Once these initial setpoints are formatted correctly, the ROS node `omni_trajctory` is initiated and the necessary publishers and subscribers are created. The flowgraph in Figure 5.9 shows the relationships between publishers and subscribers when running the trajectory publishing node.

Once the publishers and subscribers are initiated, the loop rate is set to 50 Hz. Next, the publishers begin publishing the initial position/acceleration via the `mavros/setpoint_raw/local` MAVROS topic and the and attitude setpoints via the `mavros/setpoint_attitude/attitude` and `mavros_setpoint_attitude/thrust` MAVROS topics. There are a two important notes to make here; first, before the vehicle is armed, it needs to be switched into "offboard" mode, so that it will accept setpoints from an external source. In order to do this, the FCU requires that multiple setpoints have been published at a rate higher than 20 Hz prior to arming. Thus, before arming, 50 setpoints are published to ensure that the vehicle will enter "off-

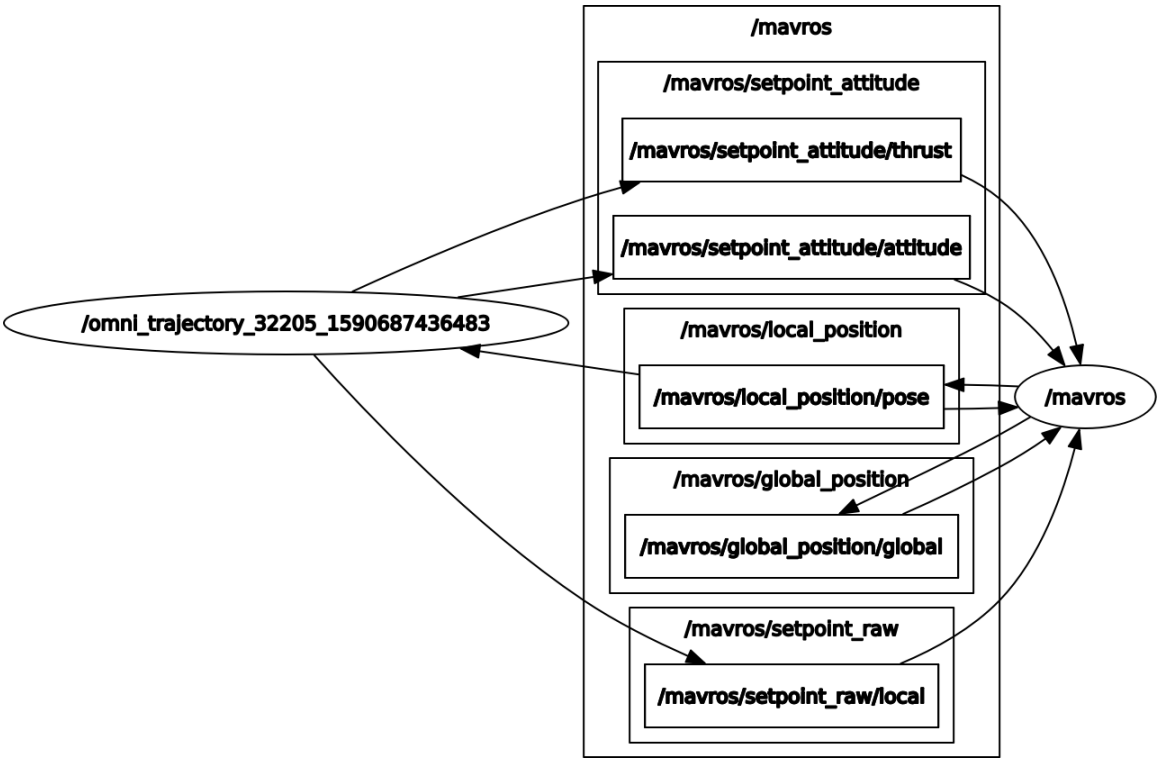


Figure 5.9: Trajectory publisher information flowgraph. This figure shows the communications between the various ROS and MAVROS topics that handle the trajectory and attitude setpoint publishing and updating.

board" mode successfully. The second note is that in order for attitude setpoints to be sent to the FCU, a thrust value setpoint must be submitted in addition to the attitude setpoint, which is a common occurrence when flying multi-rotor vehicles with planar rotor configurations. In this case, a constant thrust setpoint value of 1 is published along with the true attitude setpoints in order to satisfy the requirement, but this thrust value is never used in the attitude controller. After publishing the setpoints, a call to the `mavros/cmd/arming` service is made in an attempt to arm the vehicle. If the vehicle is successfully armed, then the `/mavros/set_mode` service is called with the "OFFBOARD" argument to set the vehicle into "offboard" mode. Once this is successful, the code enters a loop that exits only when the `omni_trajectory` node is shut down. This loop publishes the position, acceleration and attitude setpoints at 50 Hz and updates the current setpoint once the vehicle reaches the set-

point within a user-specified acceptance radius, which is nominally 0.1 m. This is achieved by subscribing to the vehicle's position via the `mavros/local_position/pose` MAVROS topic and comparing the values returned with the value of the setpoint. If the difference between the X, Y, and Z values of the setpoint and the vehicle's position are within the acceptance radius, then the next setpoint is published. This is repeated until the vehicle has flown the entire trajectory, in which case the trajectory file is looped over again starting from the second setpoint. Thus, the simulation repeats indefinitely or until it is canceled by the user.

Chapter 6

Simulation Results

6.1 Performance Validation

Once the work detailed in Chapter 5 was completed, the simulated vehicle's performance capabilities were tested. Trajectories were generated to verify capabilities in position tracking, attitude tracking, and simultaneous position and attitude tracking. When applicable, position and attitude errors are shown, and root mean square errors (RMSE) are calculated using Equation 6.1:

$$RMSE = \sqrt{\frac{1}{N}\sum_{i=1}^{N}((x_{1i} - x_{1i,des})^2 + (x_{2i} - x_{2i,des})^2 + (x_{2i} - x_{2i,des})^2)} \tag{6.1}$$

where x_1 through x_3 can represent either position or attitude values, and $x_{1,des}$ through $x_{3,des}$ are the position or attitude setpoints.

6.1.1 Position Tracking

To test position tracking, simple geometric trajectories were generated. These include a 1.5 m radius circle, a square of side length 1.5 m, as well as 45 and 90 degree inclined variants of both. In these trajectory tests, the +X-face of the vehicle is commanded to face the -X world axis, which corresponds to roll, pitch, and yaw commands of 0, 0, and 180 degrees, respectively. The simulated vs commanded trajectories and their respective error plots are shown in Figures 6.1 through 6.6.

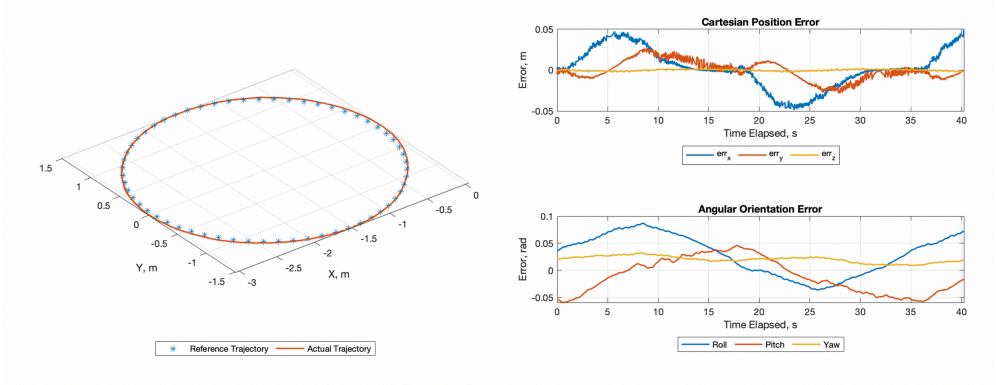


Figure 6.1: Circular planar trajectory results. The simulated vs actual trajectory is shown on the left, with the position and attitude errors on the right.

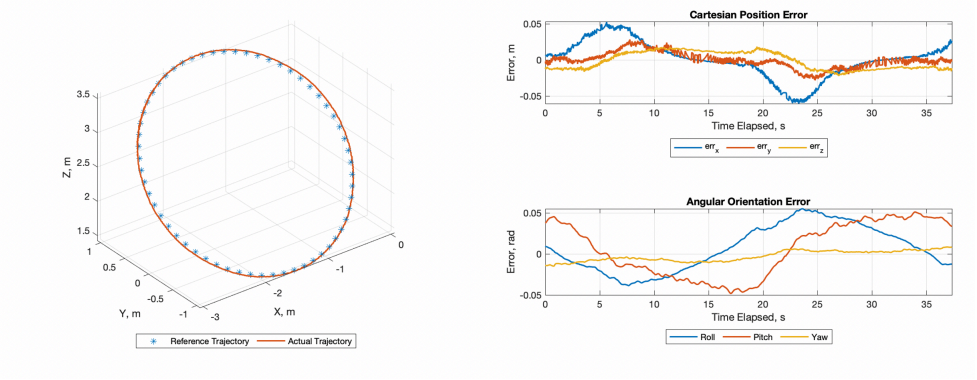


Figure 6.2: Circular 45 degree trajectory results. The simulated vs actual trajectory is shown on the left, with the position and attitude errors on the right.

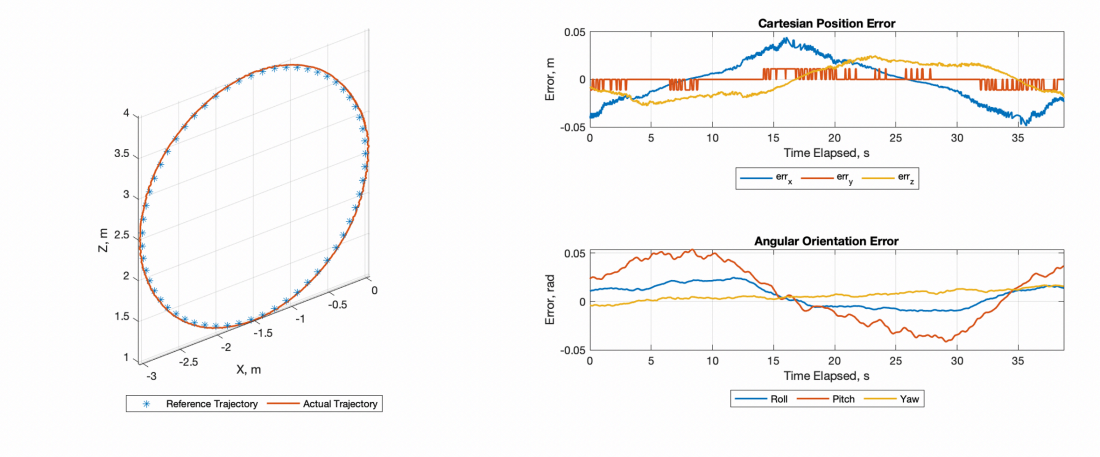


Figure 6.3: Circular 90 degree trajectory results. The simulated vs actual trajectory is shown on the left, with the position and attitude errors on the right.

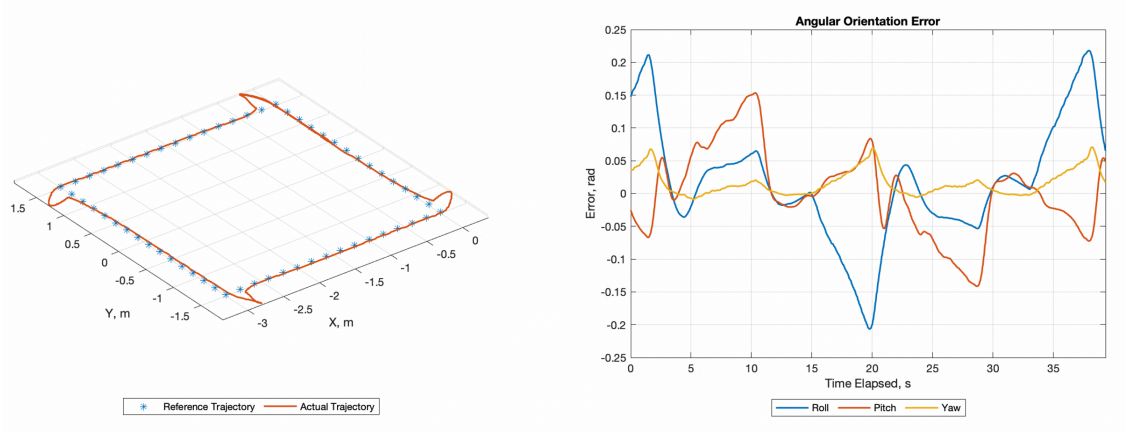


Figure 6.4: Rectangular planar trajectory results. The simulated vs actual trajectory is shown on the left, with the position and attitude errors on the right.

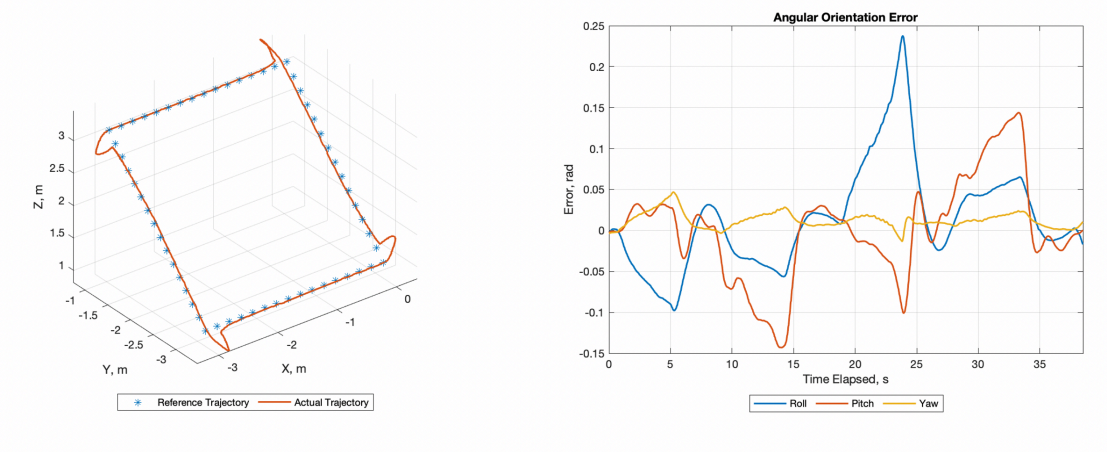


Figure 6.5: Rectangular 45 degree trajectory results. The simulated vs actual trajectory is shown on the left, with the position and attitude errors on the right.

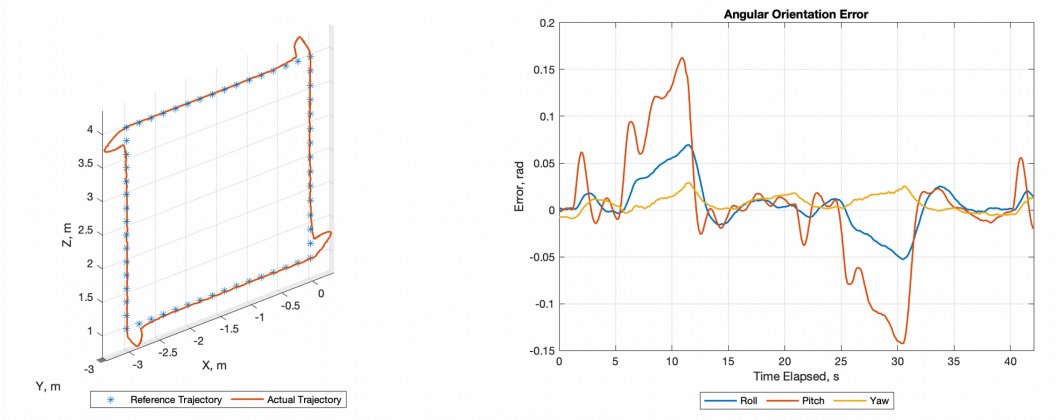


Figure 6.6: Rectangular 90 degree trajectory results. The simulated vs actual trajectory is shown on the left, with the position and attitude errors on the right.

6.1.2 Attitude Tracking

To test attitude tracking, stationary positional trajectories were generated with varying attitudes. Full 360 degree rotations were commanded about each of the vehicle's major

body axes, corresponding to full rotations in roll, pitch, and yaw. Next, a full 360 degree rotation about each body axis simultaneously was simulated. Plots of the vehicle's position and attitude during these simulations are shown in Figures 6.7 through 6.10.

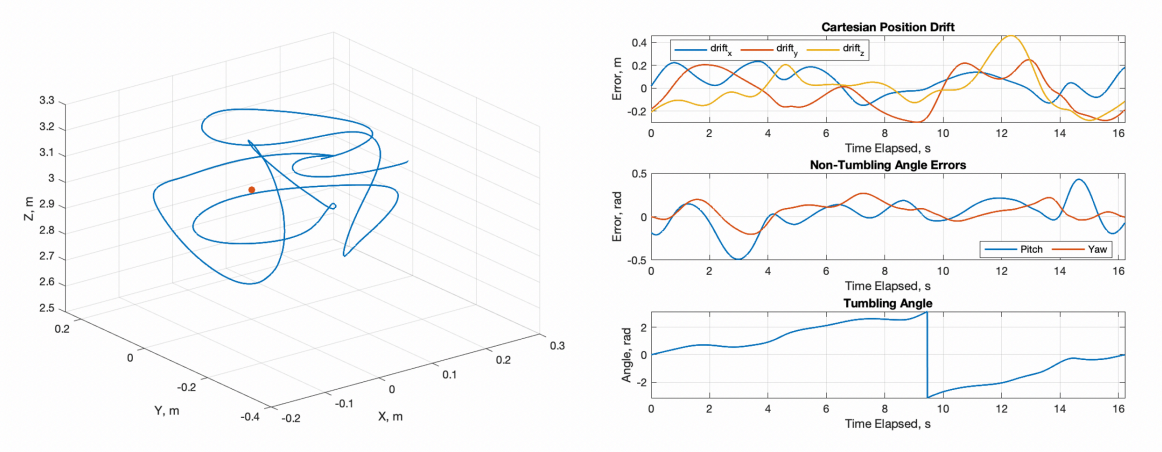


Figure 6.7: Stationary roll trajectory results. The plot on the left shows the vehicle's position about its commanded stationary point as it completes its roll maneuver, with the position and attitude errors on the right.

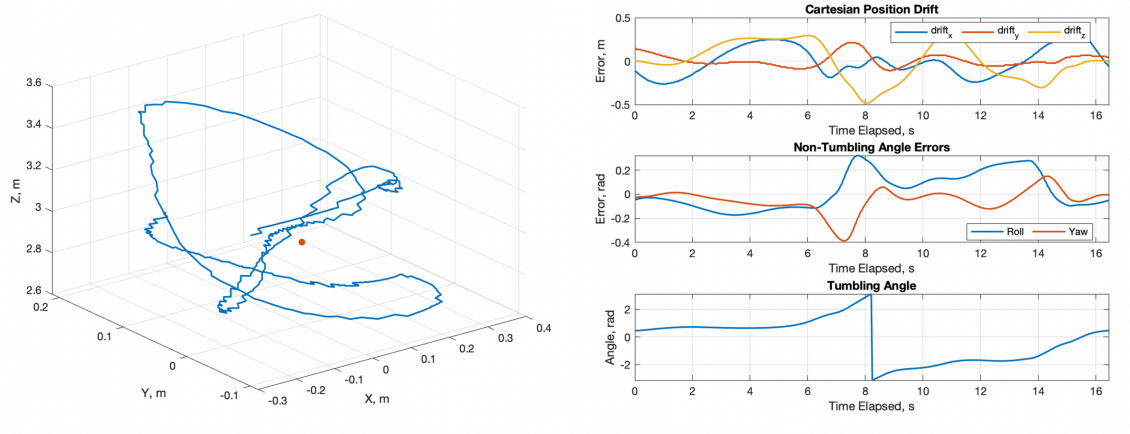


Figure 6.8: Stationary pitch trajectory results. The plot on the left shows the vehicle's position about its commanded stationary point as it completes its pitch maneuver, with the position and attitude errors on the right.

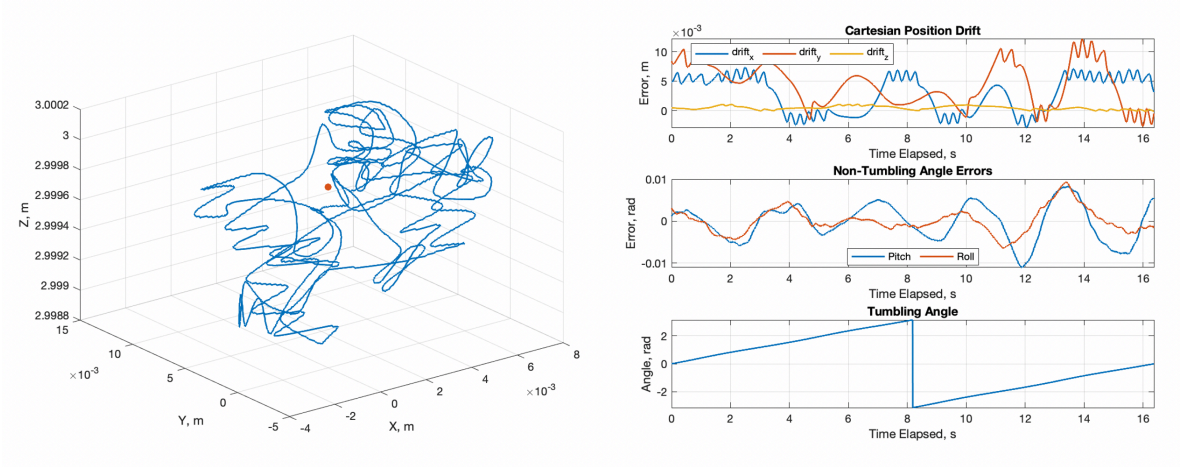


Figure 6.9: Stationary yaw trajectory results. The plot on the left shows the vehicle's position about its commanded stationary point as it completes its yaw maneuver, with the position and attitude errors on the right.

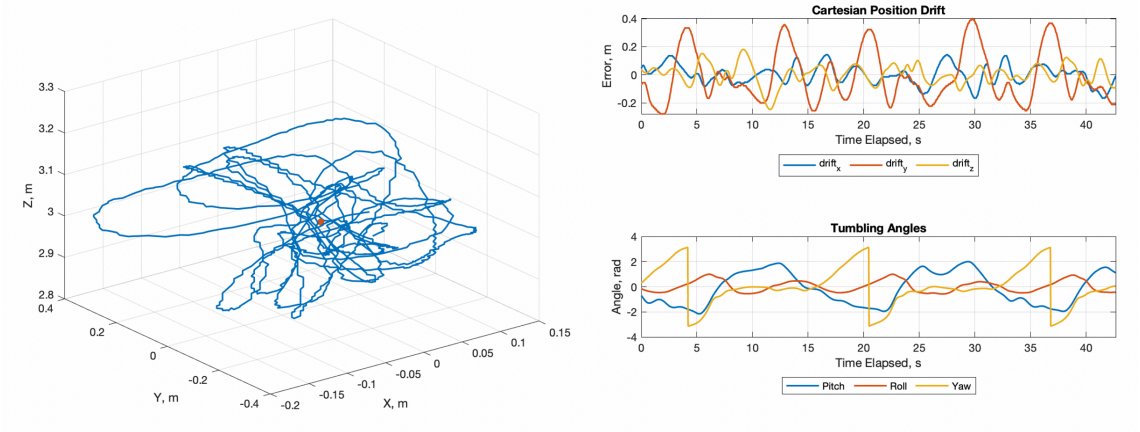


Figure 6.10: Stationary tumble trajectory results. The plot on the left shows the vehicle's position about its commanded stationary point as it completes its tumble maneuver, with the position and attitude errors on the right.

6.1.3 Simultaneous Position and Attitude Tracking

Finally, a trajectory involving simultaneous position and attitude tracking was generated. This trajectory commands 1.5 m radius circular positional trajectory while simultaneously commanding a 360 degree roll and pitch maneuver. Plots of the vehicle's trajectory, position errors and attitude values during this simulation are shown in Figure 6.11.

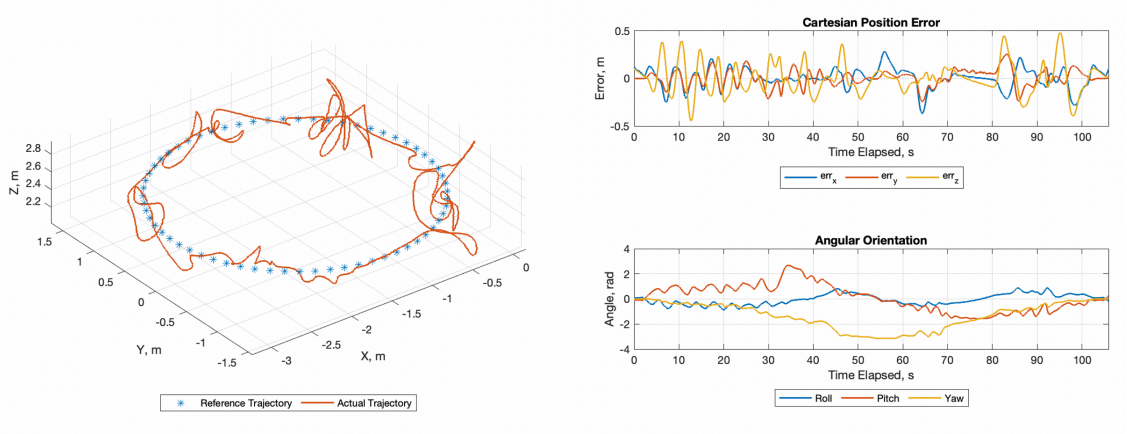


Figure 6.11: Rectangular planar trajectory results. The simulated vs actual trajectory is shown on the left, with the position errors and attitude values on the right.

6.1.4 Performance Validation Discussion

The position and attitude RMSE values for all performance validation simulations are shown in Table 6.1:

Table 6.1: Performance validation simulation RMSE values.

Trajectory:	**Position RMSE (m):**	**Attitude RMSE (rad):**
Equatorial Circle	0.0268	0.0622
45 Degree Circle	0.0301	0.0450
90 Degree Circle	0.0286	0.0345
Equatorial Rectangle	N/A	0.1142
45 Degree Rectangle	N/A	0.0906
90 Degree Rectangle	N/A	0.0659
Pitch Tumble	0.2849	0.1861
Roll Tumble	0.2845	0.2185
Yaw Tumble	0.0071	0.0052
Combined Tumble	0.2076	N/A
Circle Tumble	0.2119	N/A

The circular trajectories in Figures 6.1 through 6.3 exhibit excellent position and attitude tracking. The RMSE values for their position and attitude are small and the overall shapes of each trajectory match their respective reference trajectory quite well. Periodic deviations in the attitude are apparent as the vehicle moves throughout each trajectory, seeming to correspond to the vehicle's direction of motion at that instant; as the vehicle moves in the Y direction the roll error increases and decreases, and as it moves in the X direction the pitch error increases and decreases. This is easily discerned in Figure 6.3, where the motion is restricted to the X-Z plane, and the attitude error changes mainly in the pitch direciton. This points to a coupling that exists between the position and attitude dynamics, similar to what Brescianini and D'Andrea experienced with their first implementation of the Omnicopter before they had implemented lower-level motor control [29]. The rectangular trajectories in Figures 6.4 through 6.6 exhibit slightly worse results, though the lack of smoothness at the corners of these trajectories is the likely cause of the lower attitude tracking performance. It is important to note that the difficulty of parameterizing the rectangular trajectories in such a way that facilitates calculating the positional error led to only the attitude error being considered for these trajectories, as the error from the fixed desired attitude was easily

calculated. Besides the overshoot occurring at the corners, the positional tracking is more than acceptable. As with the circular trajectories, the coupling effect is noticable in the attitude error as the vehicle switches between accelerating along the X and Y axes; the pitch and roll errors periodically increase and decrease at an offset from each other, corresponding to the accelerations along the different sides of the rectangular trajectories, and the 90 degree rectangle contained in the X-Z plane experiences attitude error predominately in the pitch direction.

The results of the stationary attitude-based trajectories shown in Figures 6.7, 6.8, and 6.10 further exhibit this coupling between the positional and rotational dynamics, as evidenced by the positional drifts resulting in higher position RMSEs for each of these trajectories. This coupling doesn't exist when the vehicle is only commanded to yaw, as shown in Figure 6.9. As mentioned previously, the existence of this coupling is likely the result of the lack of low-level motor control implementation, which precludes the fineness of control needed to achieve the commanded torques without producing coupled forces. Even more so, the rotors' angular velocity feedback (and thus angular momenta) are unable to be accounted for within the required torque calculations, which adds uncertainty into the accuracy of the desired torque calculation. This coupling seems to be exacerbated when the desired forces are oriented toward the corners and edges of the vehicle's frame with respect to its attitude, typically corresponding to the vehicle's pitch and roll nearing values that are multiples of 45 degrees, and is at its greatest when these desired forces have elements that predominately oppose the gravitational force vector.

These possible sources of control error are supported by the results shown in Figure 6.11, where the vehicle is commanded to track a trajectory where both position and attitude are changing. The positional tracking is clearly affected by the vehicle's changes in attitude, and the attitude changes are not as smooth as could be desired, with errors occurring about the

non-changing axes.

While it is clear that the simulated vehicle in its current state experiences varying levels of error when tasked with controlling its position and attitude simultaneously, the fact that the desired trajectories are discernible from the actual trajectory points to higher possible performance once more progress is made in the implementation of lower-level motor control. It is likely that further work toward implementing the remaining aspects of the Omnicopter's controller will see most of these errors eliminated.

6.2 Two-Body Orbit Tracking

After running the performance validation simulations, tests to determine the feasibility of using the vehicle as a spacecraft dynamics simulator were conducted. Trajectories representing both circular and elliptical two-body orbits were generated, with attitude trajectories that command the vehicle to face the focus of each orbit.

6.2.1 Circular Orbits

Simulations tracking circular orbits with inclinations of 0, 30, 60 and 90 degrees, with the vehicle commanded to orient its +X-face toward the orbit's focus, were run with results shown in Figures 6.12 through 6.15.

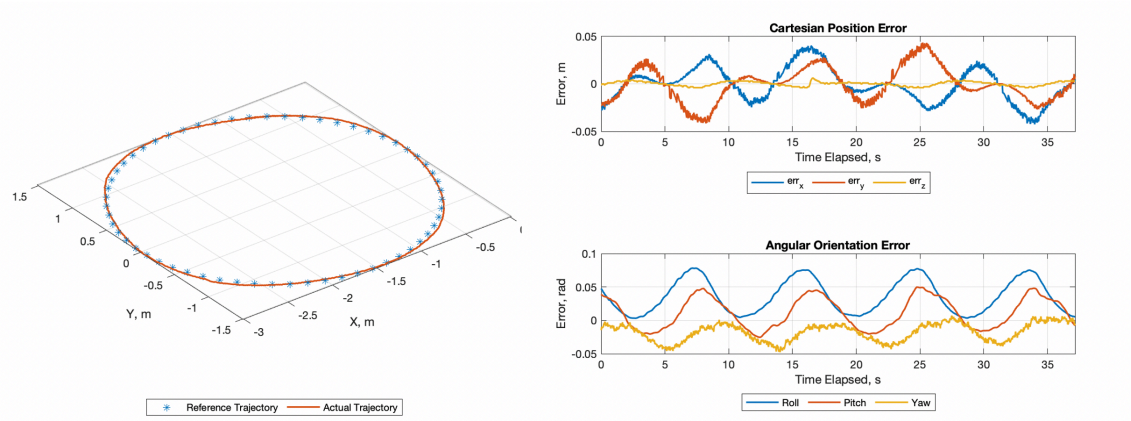


Figure 6.12: Equatorial circular orbit trajectory. The simulated vs actual trajectory is shown on the left, with the position and attitude errors on the right.

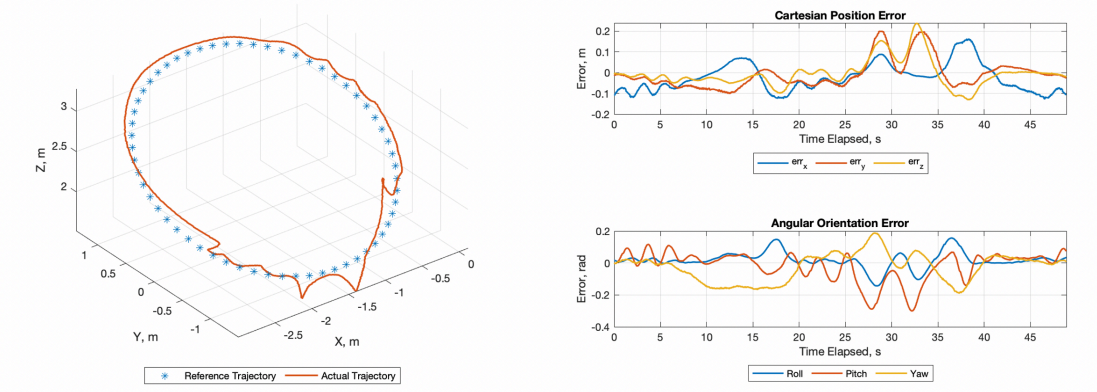


Figure 6.13: 30 degree circular orbit trajectory. The simulated vs actual trajectory is shown on the left, with the position and attitude errors on the right.

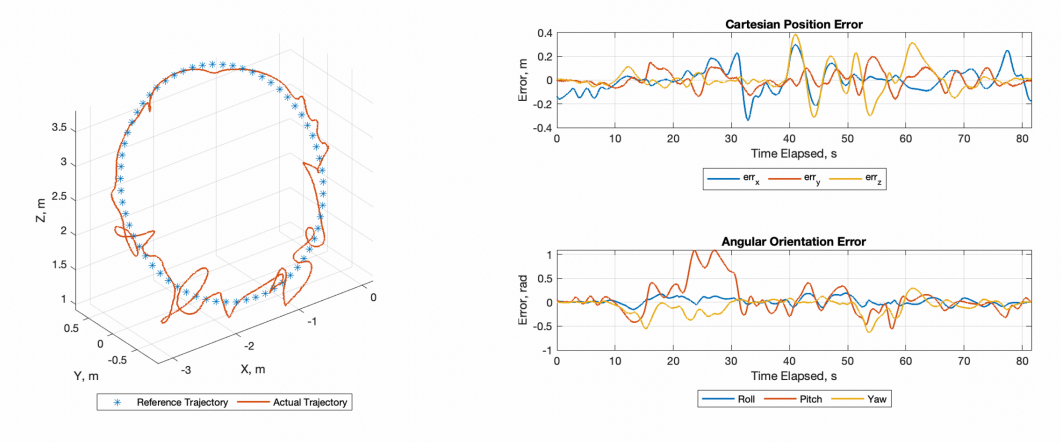


Figure 6.14: 60 degree circular orbit trajectory. The simulated vs actual trajectory is shown on the left, with the position and attitude errors on the right.

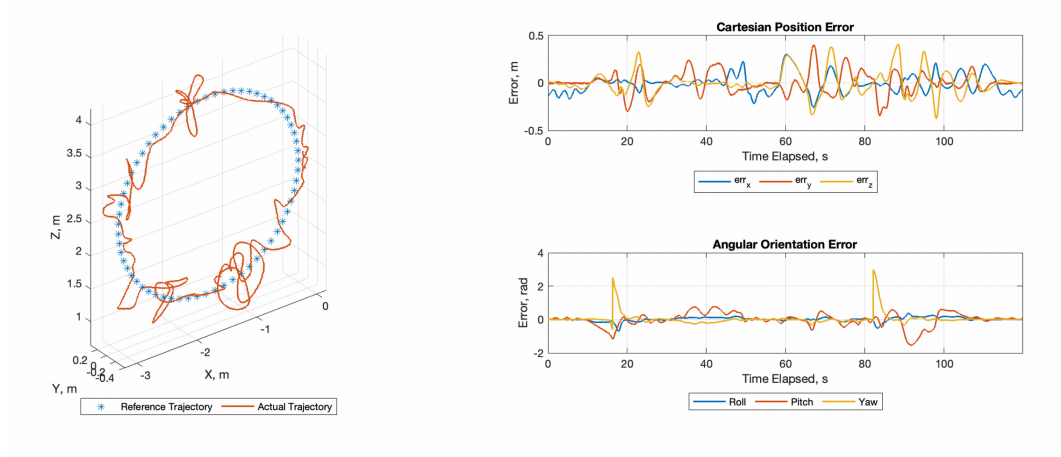


Figure 6.15: 90 degree circular orbit trajectory. The simulated vs actual trajectory is shown on the left, with the position and attitude errors on the right.

6.2.2 Elliptical Orbits

Simulations tracking elliptical orbits with inclinations of 0, 30, 60 and 90 degrees, with the vehicle commanded to orient its +X-face toward the orbit's focus, were run with results shown in Figures 6.16 through 6.19.

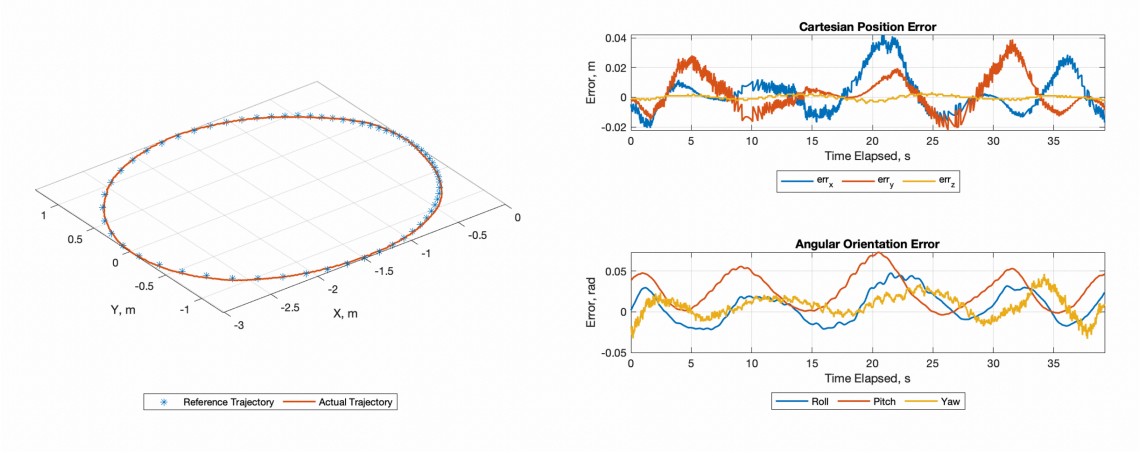


Figure 6.16: Equatorial elliptical orbit trajectory. The simulated vs actual trajectory is shown on the left, with the position and attitude errors on the right.

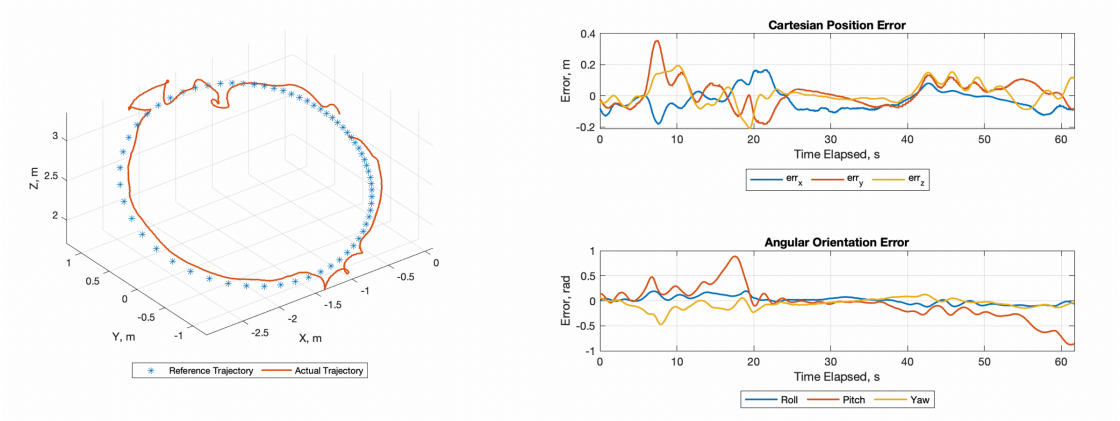


Figure 6.17: 30 degree elliptical orbit trajectory. The simulated vs actual trajectory is shown on the left, with the position and attitude errors on the right.

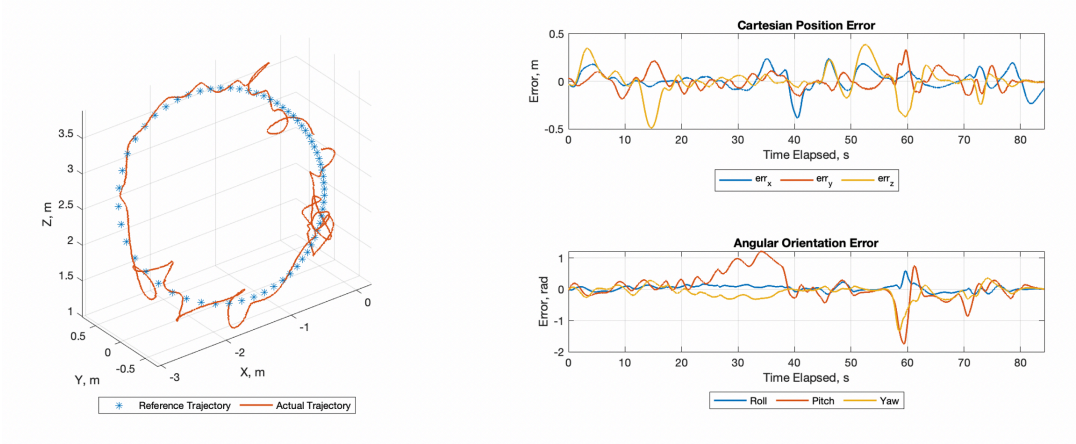


Figure 6.18: 60 degree elliptical orbit trajectory. The simulated vs actual trajectory is shown on the left, with the position and attitude errors on the right.

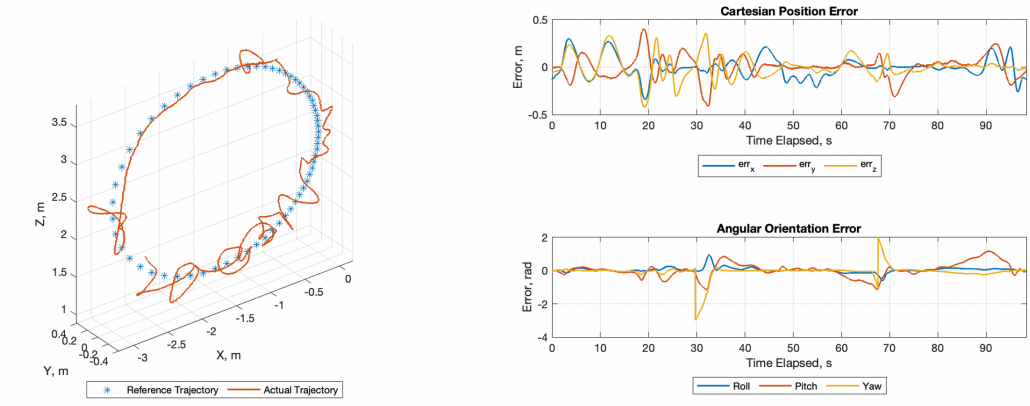


Figure 6.19: 90 degree elliptical orbit trajectory. The simulated vs actual trajectory is shown on the left, with the position and attitude errors on the right.

6.2.3 Two-Body Orbit Tracking Discussion

The position and attitude RMSE values for all performance validation simulations are shown in Table 6.2:

Table 6.2: Two-body orbit simulation RMSE values.

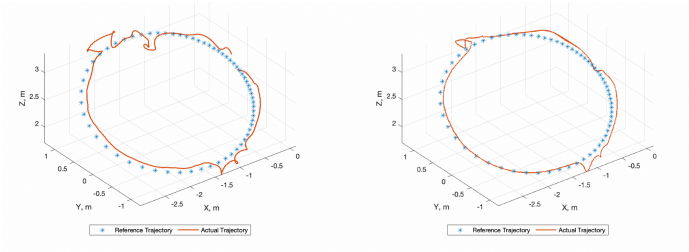
Trajectory:	**Position RMSE (m):**	**Attitude RMSE (rad):**
Equatorial Circle	0.0252	0.0556
30 Degree Circle	0.1184	0.1395
60 Degree Circle	0.1670	0.3838
90 Degree Circle	0.1956	0.5694
Equatorial Ellipse	0.0181	0.0422
30 Degree Ellipse	0.1369	0.3521
60 Degree Ellipse	0.1944	0.5254
90 Degree Ellipse	0.1998	0.6206

The results shown for the two-body orbit trajectories were not unexpected, based on the results shown for the performance validation trajectories. When the inclination of the trajectory is zero, the only commanded attitude maneuvers are in the yaw direction, which results in very high position and attitude tracking performance. As the inclination increases, attitude maneuvers occurring in the pitch and roll direction begin to occur, resulting in loss of position tracking performance as the coupling effect begins to occur when pitch and roll values approach multiples of 45 degrees. Position tracking accuracy is partially regained when the desired pitch and roll are not near multiples of 45 degrees, as evidenced in the left halves of the trajectories shown in Figures 6.13 and 6.17, but suffers again when the pitch increases towards the bottom portion of the trajectory. The 60 and 90 degree variants of both the circular and the elliptical trajectories experience large disruptions in positional tracking accuracy as a function of these attitude maneuvers, with much smaller portions of the total trajectory exhibiting accurate positional tracking.

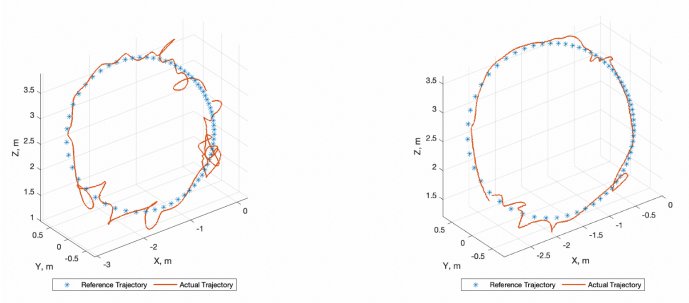
The attitude tracking performance of these trajectories varies with the inclination as well. The attitude RMSE values for each orbit increase in a fairly linear manner as the inclination increases, suggesting the larger attitude maneuvers necessary to point the vehicle towards the orbit's focus at larger inclinations become increasingly more difficult to achieve. While

the performance is mostly passable in inclined trajectories like Figures 6.13 and 6.17, the error profiles in the other inclined trajectories are notably worse. For example, in Figures 6.14 and 6.18, there are prolonged periods where the magnitude of the pitch error is much greater than zero. These periods correspond to the vehicle unsuccessfully attempting to produce the necessary torques to achieve the desired pitch angle, resulting in the vehicle later "swinging-up" or "swinging-down" rather suddenly once the desired torques become achievable, as evidenced in the oscillations in all angles occurring after these periods.

Two of the simulations, the 30 degree and 60 degree elliptical trajectories, were re-run with higher damping coefficients in the positional controllers to determine whether the positional discrepancies noted can be accounted for as underdamped harmonics. The results of these simulation re-runs are shown in Figures 6.20a and 6.20b.



(a) 30 Degree Elliptical Re-Run



(b) 60 Degree Elliptical Re-Run

Figure 6.20: Simulation re-runs with increased damping. The 30 degree and 60 degree elliptical trajectory simulations were re-run with increased damping coefficients, with the original trajectories shown on the left and the re-run trajectories shown on the right for comparison. The increased damping reduces the magnitude of the positional deviations for both cases.

Table 6.3: Damping simulation re-run RMSE values, showing values for position and attitude RMSE both before and after increasing the damping.

Trajectory:	**Position RMSE (m):**	**Attitude RMSE (rad):**
30 Degree Ellipse, before	0.1369	0.3521
30 Degree Ellipse, after	0.1129	0.6564
60 Degree Ellipse, before	0.1944	0.5254
60 Degree Ellipse, after	0.1372	0.6525

The increased damping in the positional controller reduces the magnitude of the positional deviations, as shown in Figures 6.20a and 6.20b, and evidenced by the RMSE values in Table 6.3, albeit at the expense of the attitude RMSE values. These re-runs show that increasing the damping in the positional controllers does indeed affect the errors observed, and optimally tuning this value will be explored further in the future.

As with the performance validation trajectories, it is clear that the vehicle is attempting, albeit struggling, to track the desired trajectories. While the simulated vehicle's performance is not yet fully realized, it seems that the goal of using this vehicle to simulate smallsat dynamics is feasible, and with the inclusion of direct motor controllers the vehicle will likely track trajectories representative of smallsat dynamics quite well.

Chapter 7

Conclusions and Future Work

7.1 Summary

The work done in this thesis details the progress made towards developing both physical and simulated implementations of Brescianini and D'Andrea's Omnicopter vehicle design described in [2], and evaluating its feasibility as a smallsat dynamics simulation platform. The physical vehicle is constructed similarly to Brescianini and D'Andrea's implemenation, with many of the components purchased identical to those listed in their paper. The vehicle's structure is comprised of carbon fiber rods, as in the original implementation, that connect custom-designed 3D printed corner brackets to a 3D printed central housing. This central housing contains mounting points for the flight controller, a companion computer, power distribution boards, and the flight battery. The motors are attached in the proper orientations using custom-designed 3D printed clips. While the physical implementation is not yet complete, significant progress on the software components has been made through the development of a software simulation of the Omnicopter vehicle. This simulation, using the Gazebo dynamics simulator and an SITL variant of the PX4 flight controller firmware,

integrates the Omnicopter's controller logic into PX4 modules that run directly onboard the flight controller software stack. Using a Gazebo model of the vehicle, its dynamics can be simulated allowing the software components developed as a part of this thesis to be tested and verified on multiple position and attitude trajectories. The simulated vehicle performed well on all trajectories where either position or attitude were held fixed, but coupling between the translational and rotational dynamics caused by a lack of direct motor control affected its performance during trajectories where both position and attitude maneuvers were involved. After validating its performance, the simulated vehicle was tested on a series of trajectories representative of smallsat orbital dynamics. The coupling observed in the performance validation trajectories affected the position and attitude accuracy of the simulations, but the results affirmed the vehicle's feasibility as a smallsat dynamics simulation platform.

7.2 Future Work

The most pressing future work item is to complete the physical implementation of the vehicle. First, once the flight controller is acquired, the ESCs and motors will need to be tested and their functionality verified with the flight controller. Once hardware functionality is assured, communication procedures between the companion computer and the flight controller will need to be developed. Finally, the flight controller, the companion computer, the battery, and the ESCs will need to be mounted and secured to the airframe. Once all necessary components have been secured to the vehicle, component tests will be re-conducted.

Alongside completing the physical vehicle, the software components of the vehicle need further development. The completion of the physical vehicle will necessitate the splitting of the software development into vehicle-focused and simulation-focused branches, as though the architecture of the firmware environment is the same between the simulated and physical

software implementations there are nuances and code differences between them that will require having two separate iterations of the flight controller firmware. With regards to vehicle-focused development, the direct motor controller will need to be integrated as a PX4 module and instances of motor angular velocity feedback in other control loops will need to be included. In the context of the physical vehicle this will be straightforward, as it has already been verified that the motor angular velocities can be measured using the telemetry included with the DShot ESCs. Therefore, the only obstacle in this regard is writing the code for the module and testing it on the vehicle. Additionally, it would be beneficial to run the firmware in a hardware-in-the-loop testing environment, whereby the software and all computations run on the physical flight controller and the outputs are directed into the Gazebo simulation, to verify that the modules developed run as expected on the target flight hardware. With regards to simulation-focused development, the same integration of direct motor control and motor angular velocity feedback will need to be implemented. This is less straightforward, as MAVROS and Gazebo plugins will need to be developed in order to extract the individual angular velocities of each motor. Additionally, a method for mapping the outputs of the motor controller to the inputs of the Gazebo motors will need to be determined, as the motor controller outputs individual motor voltage values, which are not usable in the context of the simulation.

There are multiple fine details included in the original Omnicopter paper that will need to be implemented to ensure the performance of the vehicle once the above work is complete. While the current implementation of the controller loops utilize physical properties listed in Brescianini and D'Andrea's paper for their implementation of the Omnicopter, it will be necessary to re-measure these properties for our implementation of the vehicle. Properties to measure will include vehicle mass and moments of inertia, rotor mass and moments of inertia, rotor aerodynamic properties, and motor dynamic models. Rotor aerodynamic properties,

such as thrust and torque coefficients, can be determined via load cell tests and dynamic system modeling, and motor dynamic models can be developed using system ID techniques in a manner similar to Brescianini and D'Andrea's implementation. Additionally, Brescianini and D'Andrea determined factors with which to correct the commanded motor angular velocities to account for loss of rotor thrust and torque performance as a result of aerodynamic interference, which will be necessary to ensure the vehicle responds appropriately to the commanded inputs.

Bibliography

[1] NASA, "CubeSat Launch Initiative," *NASA*, 2020. URL https://www.nasa.gov/content/about-cubesat-launch-initiative.

[2] Brescianini, D., and D'Andrea, R., "An Omni-Directional Multirotor Vehicle," *Mechatronics*, Vol. 55, 2018, pp. 76 – 93. https://doi.org/https://doi.org/10.1016/j.mechatronics.2018.08.005, URL http://www.sciencedirect.com/science/article/pii/S0957415818301314.

[3] Mohan, S., Saenz-Otero, A., Nolet, S., Miller, D. W., and Sell, S. T., "SPHERES flight operations testing and execution," 2009.

[4] NASA, "Zero Gravity Research Facility," *NASA Technical Report Server*, 2020. URL https://www1.grc.nasa.gov/facilities/zero-g/#experimental-drop-vehicle.

[5] Corporation, Z. G. R., 2020. URL https://www.gozerog.com/index.cfm?fuseaction=Research_Programs.welcome.

[6] NASA, "Sonny Carter Training Facility: The Neutral Buoyancy Laboratory," *NASA Johnson Space Center*, 2020. URL https://www.nasa.gov/centers/johnson/pdf/167748main_FS_NBL508c.pdf.

[7] NASA, "Air Bearing Floor," *NASA Johnson Space Center*, 2020. URL

https://www.nasa.gov/centers/johnson/engineering/integrated_environments/air_bearing_floor/index.html.

[8] Wilde, M., Kaplinger, B., Go, T., Gutierrez, H., and Kirk, D., "ORION: A simulation environment for spacecraft formation flight, capture, and orbital robotics," *2016 IEEE Aerospace Conference*, 2016, pp. 1–14. https://doi.org/10.1109/AERO.2016.7500575.

[9] Thomas, D., Wolosik, A. T., and Black, J., "CubeSat Attitude Control Simulator Design," 2019. https://doi.org/10.2514/6.2018-1391, URL https://arc.aiaa.org/doi/abs/10.2514/6.2018-1391.

[10] Gargioni, G., Peterson, M., Persons, J. B., Schroeder, K., and Black, J., "A Full Distributed Multipurpose Autonomous Flight System Using 3D Position Tracking and ROS," 2019, pp. 1458–1466. https://doi.org/10.1109/ICUAS.2019.8798163.

[11] Crowther, B., Lanzon, A., Maya-Gonzalez, M., and Langkamp, D., "Kinematic Analysis and Control Design for a Nonplanar Multirotor Vehicle," *Journal of Guidance, Control, and Dynamics*, Vol. 34, No. 4, 2011, pp. 1157–1171. https://doi.org/10.2514/1.51186, URL https://doi.org/10.2514/1.51186.

[12] Jiang, G., and Voyles, R., "Hexrotor UAV platform enabling dextrous interaction with structures-flight test," *2013 IEEE International Symposium on Safety, Security, and Rescue Robotics (SSRR)*, 2013, pp. 1–6. https://doi.org/10.1109/SSRR.2013.6719377.

[13] Kaufman, E., Caldwell, K., Lee, D., and Lee, T., "Design and development of a free-floating hexrotor UAV for 6-DOF maneuvers," *2014 IEEE Aerospace Conference*, 2014, pp. 1–10. https://doi.org/10.1109/AERO.2014.6836427.

[14] Rajappa, S., Ryll, M., Bülthoff, H., and Franchi, A., "Modeling, Control and Design

Optimization for a Fully-actuated Hexarotor Aerial Vehicle with Tilted Propellers," 2015. https://doi.org/10.1109/ICRA.2015.7139759.

[15] Nikou, A., Gavridis, G. C., and Kyriakopoulos, K. J., "Mechanical design, modelling and control of a novel aerial manipulator," *2015 IEEE International Conference on Robotics and Automation (ICRA)*, 2015, pp. 4698–4703. https://doi.org/10.1109/ICRA.2015.7139851.

[16] Park, S., Her, J., Kim, J., and Lee, D., "Design, modeling and control of omni-directional aerial robot," *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2016, pp. 1570–1575. https://doi.org/10.1109/IROS.2016.7759254.

[17] Ryll, M., Bülthoff, H. H., and Giordano, P. R., "Modeling and control of a quadrotor UAV with tilting propellers," *2012 IEEE International Conference on Robotics and Automation*, 2012, pp. 4606–4613. https://doi.org/10.1109/ICRA.2012.6225129.

[18] Segui-Gasco, P., Al-Rihani, Y., Shin, H., and Savvaris, A., "A novel actuation concept for a multi rotor UAV," *2013 International Conference on Unmanned Aircraft Systems (ICUAS)*, 2013, pp. 373–382. https://doi.org/10.1109/ICUAS.2013.6564711.

[19] Long, Y., and Cappelleri, D. J., "Linear control design, allocation, and implementation for the Omnicopter MAV," *2013 IEEE International Conference on Robotics and Automation*, 2013, pp. 289–294.

[20] Ryll, M., Bicego, D., and Franchi, A., "Modeling and control of FAST-Hex: A fully-actuated by synchronized-tilting hexarotor," *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2016, pp. 1689–1694. https://doi.org/10.1109/IROS.2016.7759271.

[21] McCormick, B. W., *Aerodynamics, Aeronautics, and Flight Mechanics*, 2nd ed., Wiley New York, 1995.

[22] Howard, D. A., and Koenig, N., "Gazebo Robotics Simulator," , ???? URL http://gazebosim.org/.

[23] Quigley, M., Conley, K., Gerkey, B., Faust, J., Foote, T., Leibs, J., Wheeler, R., and Ng, A., "ROS: an open-source Robot Operating System," 2009.

[24] Dronecode, "PX4 Open Source Autopilot," , 2020. URL https://px4.io/.

[25] Meier, L., "Micro Air Vehicle Link," , 2020. URL https://mavlink.io/en/.

[26] Holybro, "Pixhawk 4 Mini Flight Controller," , 2020. URL https://docs.px4.io/master/en/flight_controller/pixhawk4.html.

[27] Pixhawk, "DShot ESC Protocol," , 2020. URL https://docs.px4.io/master/en/peripherals/dshot.html.

[28] Pixhawk, "Pixracer Flight Controller," , 2020. URL https://docs.px4.io/v1.9.0/en/flight_controller/pixracer.html.

[29] Brescianini, D., and D'Andrea, R., "Design, modeling and control of an omni-directional aerial vehicle," *2016 IEEE International Conference on Robotics and Automation (ICRA)*, 2016, pp. 3261–3266.