An Intelligent Drone Testbed for Control Systems and Verification

Amin Yahyaabadi

A Thesis submitted to the Faculty of Graduate Studies of

The University of Manitoba

in partial fulfilment of the requirements of the degree of

Master of Science

Department of Mechanical Engineering

University of Manitoba

Copyright ©2021 by Amin Yahyaabadi

Contents

1 Introduction **11**

1.1 Motivations 12

1.1.1 The need for the development of new technologies . . . 12

1.1.2 The need for precision system identification of drones . 14

1.1.3 Challenges with custom drone software 14

1.2 Hypotheses 15

1.3 Literature Review 15

1.3.1 Testbed for the development of new technologies 15

1.3.2 System-identification of drones 17

1.3.3 Custom drone software using automatic code generation 22

1.4 Contributions 23

1.4.1 Testbed for the development of new technologies 23

1.4.2 System-identification of drones 24

1.4.3 Custom drone software using automatic code generation 25

1.5 Outline 26

2 Equations of the Motion **27**

2.1 Kinematics 27

2.2 Dynamics . . . 33
2.3 Aerodynamics . . . 36
2.3.1 Aerodynamics model . . . 36
2.4 State-space equations . . . 40
2.5 Quadrotor aerodynamics . . . 41

3 Closed-loop Control 43
3.1 Control problem definition . . . 44
3.2 Control Commands . . . 45
3.3 Controllers . . . 53
3.3.1 Height Controller . . . 54
3.3.2 Yaw Controller . . . 56
3.3.3 x and y Control . . . 57
3.3.4 Landing logic . . . 60
3.4 Closed-loop control results . . . 60

4 Feed-forward Drone Control 70
4.1 Aerodynamic force (F_{aero}) . . . 72
4.2 Thrust force (F_T) . . . 74
4.3 Feed-forward controller commands . . . 77

5 System Identification 80
5.1 Graybox model - Metaheuristic algorithm . . . 82
5.1.1 Model parameters . . . 83
5.2 Particle Swarm Optimization . . . 84
5.2.1 Population initialization . . . 85
5.2.2 Cost function calculation . . . 85

5.2.3 Outliers removal 89
5.2.4 Finding the best answers 89
5.2.5 Updating the velocities 89
5.2.6 Updating the locations 90
5.2.7 Stopping criteria 90
5.3 Algorithm performance 91

6 Software **94**
6.1 Onboard software architecture 97
6.1.1 Graphical User Interface (GUI) 99
6.1.2 Computer command and data transceiver 100
6.1.3 Onboard sensors 101
6.1.4 Signal processor 101
6.1.5 Estimator 101
6.1.6 Computer command receiver 102
6.1.7 Reference generator 102
6.1.8 Data logger 103
6.1.9 Data sender 103
6.1.10 Drone dynamics commentator and virtual dynamics engine 103
6.1.11 Candidate controller 104
6.2 Programming for code generation 104
6.3 Benchmarking auto-code generation 109
6.4 Conclusion 113

7 Results and Conclusion **114**

7.1 Feed-forward control results 114

7.1.1 Linear feed-forward injection gain 115

7.1.2 Sigmoid feed-forward injection gain 126

7.2 Conclusion 133

7.3 Future work 134

A Parrot Mambo physical properties **136**

B The experimental setup **138**

C Handwritten C code used for benchmarking **142**

References **151**

Abstract

This research develops a drone testbed that enables the rapid development, validation, and qualification of novel spacecraft and drone control algorithms, control hardware, and remote sensing technologies. The drone testbed facilitates advanced navigation and control research by emulating a multitude of dynamic environments, which provides an easily accessible framework for using new technologies in industrial applications. This research introduces an intelligent system identification method to determine the dynamics and aerodynamic parameters of the drone using Particle Swarm Optimization (PSO), a metaheuristic algorithm. This method provides the knowledge of the system's dynamics that is needed for the environment emulation.

List of Tables

3.1 Controller gains 61

6.1 Traditional embedded programming vs auto-code generation . 96

A.1 Parrot Mambo physical properties 136

List of Figures

1.1 Technology Readiness Level 13
1.2 Drone testbed block diagram 24

2.1 Drone frame set and motor configuration 28

3.1 Drone plant 43
3.2 System block diagram. 44
3.3 Translation of the drone in z direction. 46
3.4 Yaw rotation of the drone. 47
3.5 Roll rotation of the drone. 48
3.6 Pitch rotation of the drone. 48
3.7 The closed-loop controller. 55
3.8 Height Controller 56
3.9 Yaw Controller 56
3.10 Cascade controllers 57
3.11 Desired roll and pitch controllers 58
3.12 Roll and pitch controllers 59
3.13 Parrot Mambo altitude (z) in a hover flight with the desired height of 1 m 62

3.14 Parrot Mambo error in the z direction in a hover flight with the desired height of 1 m 63
3.15 Parrot Mambo velocity in the z direction in a hover flight with the desired height of 1 m 63
3.16 The closed-loop command sent to motor 3 in a hover flight with the desired height of 1 m. 64
3.17 The closed-loop command sent to motor 4 in a hover flight with the desired height of 1 m. 65
3.18 Parrot Mambo altitude (z) in a hover flight with the variable desired height of 1 m to 2 m. 66
3.19 Parrot Mambo velocity in the z direction in a hover flight with the variable desired height of 1 m to 2 m. 67
3.20 The closed-loop command sent to motor 3 in a hover flight with the variable desired height of 1 m to 2 m. 68
3.21 The closed-loop command sent to motor 4 in a hover flight with the variable desired height of 1 m to 2 m. 69

4.1 The feed-forward controller and the closed-loop controller. In this figure, the drone plant includes the sensors and the on-board estimator. 71

5.1 System Identification 82
5.2 Convergence of PSO cost function to zero 92
5.3 The performance of the mathematical model when PSO is used to estimate the thrust coefficients. 93

6.1 Auto code generation procedure 97

6.2 Onboard software architecture 98
6.3 The flight control graphical user interface 100
6.4 A sample Simulink file used to demonstrate code-generation . 110

7.1 The feed-forward injection gain curves 115
7.2 The altitude of the drone in a virtual environment with 25% reduced gravity 117
7.3 The total command sent to motor 3 in a virtual environment with 25% reduced gravity 118
7.4 The closed-loop command sent to motor 3 in a virtual environment with 25% reduced gravity 120
7.5 The closed-loop command sent to motor 4 in a virtual environment with 25% reduced gravity 121
7.6 The altitude of the drone in a virtual environment with 64% reduced gravity 122
7.7 The closed-loop command sent to motor 3 in a virtual environment with 64% reduced gravity 123
7.8 The total command sent to motor 3 in a virtual environment with 64% reduced gravity 124
7.9 The effect of correct coefficients of thrust on a lower closed-loop command in the feed-forward mode 125
7.10 The feed-forward control performance measure (zoomed-in) . 126
7.11 The sigmoid feed-forward injection gain over time 128
7.12 The altitude of the drone in a virtual environment with 49% reduced gravity and a sigmoid injection gain 129

7.13 The velocity of the drone in a virtual environment with 49% reduced gravity and a sigmoid injection gain 130
7.14 The closed-loop command sent to motor 3 in a virtual environment with 49% reduced gravity and a sigmoid injection gain 131
7.15 The closed-loop command sent to motor 4 in a virtual environment with 49% reduced gravity and a sigmoid injection gain 132

B.1 The drone testbed 138
B.2 Parrot Mambo 139
B.3 Vicon Cameras 140
B.4 DJI F550 hexacopter 141

Chapter 1

Introduction

This thesis introduces and develops a drone testbed that allows us to easily develop and validate new spacecraft and drone hardware and software algorithms. By emulating different virtual dynamics or environments, this testbed provides a method for testing and qualifying new technologies and makes the research in control and navigation fields accessible, which eventually allows the use of these novel technologies in industrial applications.

To achieve this goal, knowledge of the dynamics of the system is needed. This research develops an intelligent system identification method that can model the dynamics and aerodynamics of the drone using metaheuristic algorithms such as Particle Swarm Optimization (PSO).

The drone testbed consists of a copter-style drone flying in an indoor environment with Vicon motion capture cameras and computer interfaces, which allow measuring, testing, and data analysis capabilities. The flight and ground software is implemented using automatic code generation, which allows hardware and processor in the loop software development.

1.1 Motivations

1.1.1 The need for the development of new technologies

Applying new technologies has always been challenging and expensive in the space industry because of the number of qualification tests needed [1, 2]. As a result, there has always been some resistance to implementing novel methods and technologies even though they could dramatically improve the space industry, reduce costs, and develop development times. The technology readiness level is a measure that demonstrates the amount of the tests and verification that new technology has passed, which indicates the readiness of the technology for practical usage (Figure 1.1). The rigidity of this measure has resulted in a considerable time lag between the latest science and technology advancements and what is used in the industry, and therefore this has caused several challenges to remain unaddressed [3]. One of these challenges caused by using dated control algorithms is the potential instability of the control algorithms in emergency situations when inputs to the system or system variables are out of range, potentially resulting in irrecoverable disasters such as a crash landing and mission failure. Another challenge arises when the traditional control and estimation algorithms are used with fixed parameters resulting in non-optimized operations, particularly when the system's dynamics change over time, either due to normal ageing or the system's failures. The inability of the traditional control algorithms to adapt to and predict the dynamic changes requires the designers to verify that a single or a collection of fixed-parameter controllers can be suitable for the entire mission. This forces the engineers to spend excessive amounts of time

and money to predict and simulate all the possible combinations of failures and changes to the system over its entire lifetime.

Through this testbed, new technologies can be tested and qualified in the virtual dynamic environments created by a drone. This hardware-in-the-loop testbed allows these technologies to reach a technology readiness level of 6-7 (as shown with arrows in Figure 1.1). Therefore, the qualification activities would be faster and less expensive.

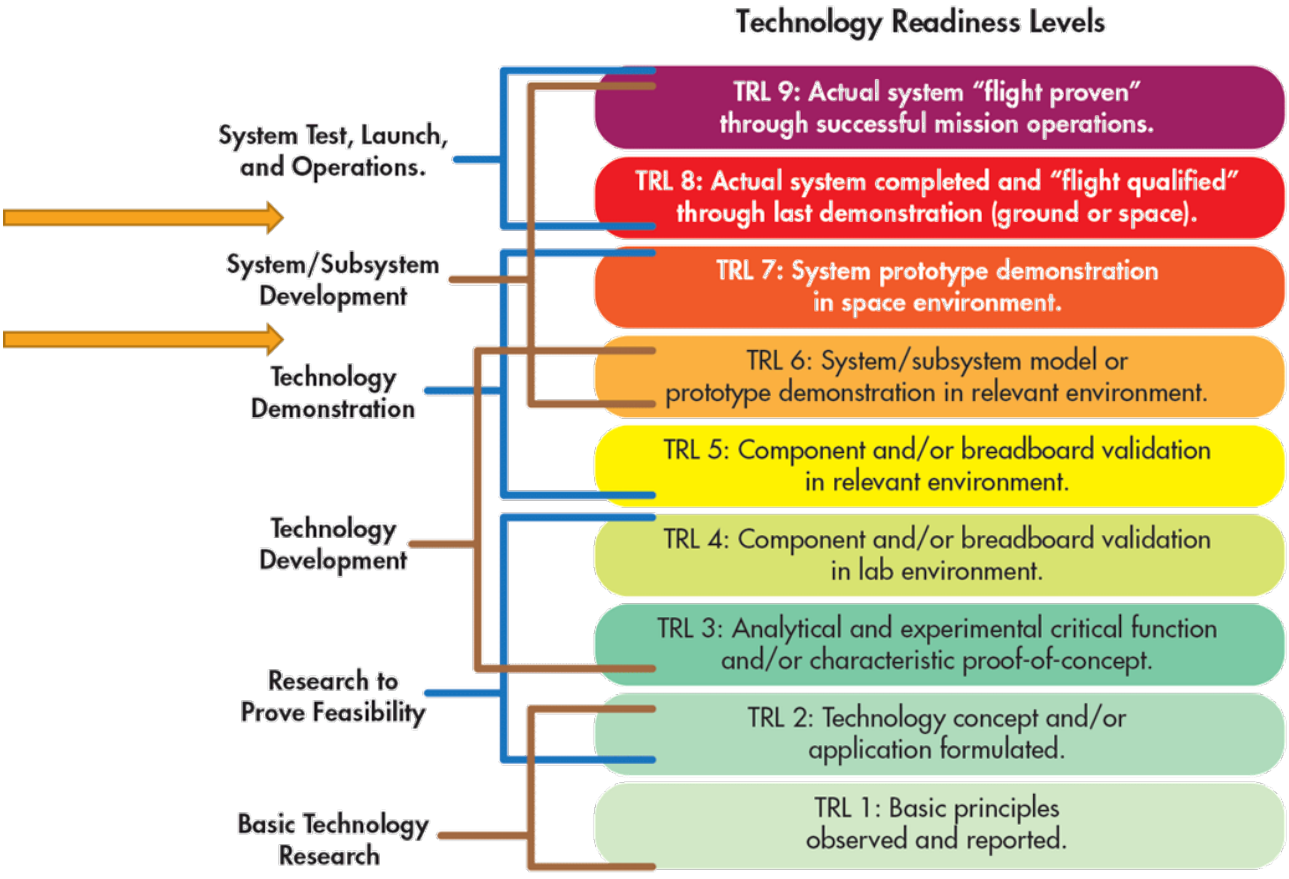

Figure 1.1: Technology Readiness Level

1.1.2 The need for precision system identification of drones

For simulating virtual dynamic environments using drones, a feed-forward controller that compensates for the dynamics of the drone should be implemented. That means that a very accurate model of the drone's dynamics and aerodynamics is needed so that I can reverse and compensate for it. In contrast to a closed-loop controller that uses real-time data from the sensors to measure the states and adjust the motor speeds to achieve the desired goal, feed-forward controllers should only rely on using the available information about the drone's dynamics and environment.

In practice, the actual UAV dynamics are very complicated, and as discussed in Section §1.3, different parameters and effects such as the misalignment of the thrust vectors, propeller pitches, deviation from the hover pose, the location of the center of mass, motor torque constants, minor variations in voltage, etc. will have significant effects on the system. Because many factors affect the drone's dynamics, classical system identification methods will either fail to find a dynamics model or only give a sub-optimal answer [4]. For this reason, using an intelligent data-driven approach is necessary to model the drone.

1.1.3 Challenges with custom drone software

Fast code prototyping and actual hardware implementation are considered contrary to each other. This makes taking the idea from the prototype to the implementation in the embedded software a complex challenge. Following the traditional embedded programming practices makes experimenting with

algorithms and reusing the code for different hardware or missions very time-consuming and out of reach of researchers.

1.2 Hypotheses

The hypotheses are broken down into the following:

1. **Testbed for the development of new technologies**: Drones can be used as a testbed to develop new technologies. To achieve this, virtual dynamics can be emulated using feed-forward control. In this research, the emulation of a reduced gravity environment is chosen.

2. **System Identification of drones:** It is possible to identify the drone dynamics parameters based on the flight data with minimal direct measuring [1]. The PSO algorithm can be used to estimate the system parameters based on data.

3. **Custom drone software using automatic code generation:** Performant custom flight and ground software can be automatically generated from a high-level language.

1.3 Literature Review

1.3.1 Testbed for the development of new technologies

Because of the need for the ground recreation of the space environment and a spacecraft's motion, several testbeds have been made. These testbeds are

[1] only for the mass and airframe dimensions

mainly in the form of small air-bearing tables usually used for small or microsatellites or in huge sizes aimed at large satellites at NASA [5].

In [6], Ferguson used a 1-G air-bearing floor at MIT as a microgravity simulator to provide contrasting dynamic and gravitational environments. Air-bearing simulators create a film of pressurized gas between the table and the air-bearing vehicle. They can be used to simulate motions usually with 1 DOF[2] (Flat air-bearings), 3 DOF (hemispherical air-bearings) [7], 4 DOF (a combination of a hemispherical air-bearing with an air-bearing rail system) [8], and 5 DOF (a combination of a hemispherical air-bearing with a planar air-bearing) [9].

The NASA Rendezvous Docking Simulator [10, 11, 12, 13] at Langley Research Center in Hampton, Virginia, was a proximity maneuver simulator. This huge-size simulator was first designed for the simulation of docking between the Gemini spacecraft and the Agena target vehicle. This facility also simulates docking between the Apollo Command and Service Module and a Lunar Module.

The Formation Control Testbed at the Jet Propulsion Laboratory was used for the Terrestrial Planet Finder mission to verify and test the control algorithms in formation flight [14].

The Space Systems Laboratory (SSL) Synchronized Position Hold Engage and Reorient Experimental Satellites (SPHERES) testbed [15] provides a testing facility for the Distributed Satellite Systems (DSS) and control algorithms for formation flight and docking.

As discussed in 1.1.1, these facilities are very costly, making them inac-

[2]Degree Of Freedom

cessible and out of reach. The testbed developed as part of this research project provides a hardware-in-the-loop environment for testing and verification with a fraction of the cost and maintenance effort needed for the existing technologies. Using this testbed, the development, test, and verification of new technologies become accessible to a broader range of researchers and engineers, making it possible to address the challenges that exist in the older technologies.

1.3.2 System-identification of drones

Different research has been done to study the system identification methods for modelling dynamics and aerodynamics. In the following, some of the different approaches are discussed:

Direct measurement and manual tweaking

Bresciani [16] studied the calculation and measurement of different parameters of a quadrotor model. They considered the model of a DC-motor, the gearbox, and the propeller to reach better accuracy. Nonami [17] developed a helicopter model and measured some of its parameters, and for the rest of the parameters, they did manual tweaking until the model and actual experiment results matched each other. Elsamanty in [18] described a methodology to identify all of the parameters of a quadrotor system, including the structure parameters and rotor assembly parameters. They built three simple test rigs to determine the relationship between the motor input Pulse Width Modulation (PWM) signal and the angular velocity, the thrust force, and the drag moment of the rotors.

While direct measurements could be helpful in particular cases, the cost, effort, and facilities needed for this make it out of reach for most researchers [4], which brings us to the next section.

System identification methods

In [19], Derafa developed a drone model and considered different effects such as aerodynamic friction and gyroscopic effects, and then used system identification to estimate parameters. Stanculeanu [20] used a new approach to identify the quadrotor dynamic model using a black box and tried to address problems that arise in determining the closed-loop behaviour. Stanculeanu also offered a technical solution for overcoming the correlation between the input noise present in the output.

Abas, in [21, 22], identified an unknown parameter of the quadrotor using a state estimation method with the implementation of an Unscented Kalman Filter (UKF). Beard in [23] discussed different sensor measurement equations and state estimation using a Kalman filter for a quadrotor drone. In [24], Lyu estimated the parameters of the thrust, drag force, torque, rolling moment, and pitching moment via the Kalman filter. Global Positioning System (GPS) and inertial sensors were used as measurements.

In [25], Sa described the system identification, estimation, and control of translational motion and the heading angle for a cost-effective open-source quadcopter. In [26], Rich presented a comprehensive model considering different aerodynamic effects and found different parameters using system identification.

In [27], Chovancová presented three drone models, including the nonlinear

model, a model described in quaternions, and a near-hover model. He investigated the parameters that can be calculated or experimentally identified such as arm length, the total mass of the quadrotor, inertia matrix, friction coefficients, thrust coefficient, and drag coefficient. He also described that the experimental identification of an actuator is usually required.

Zhang in [28] gave a tutorial of the platform configuration, methodology of modelling, comprehensive nonlinear model, the aerodynamic effects, and model identification for a quadrotor. Bisheban in [29] presented a computational framework to identify the impcat of wind on the dynamics of a quadrotor. In [30], Filatov was concerned with the thrust generation subsystem for small quadrotors and identifying its parameters using a black box. In [31], Angarita presented an initial analysis of methods used to generate and evaluate a range of model structures that best define a generic quadrotor. Sung [32] made the system identification of a 3DR X8+ aircraft via frequency-domain system identification techniques.

In [33], the aerial manipulator model was first derived analytically using the Newton-Euler formulation for the quadrotor and the Recursive Newton-Euler formulation for the manipulator. The simulation data was then used for system identification of the aerial manipulator. AutoRegressive with exogenous inputs (ARX) models were obtained from the system identification for linear accelerations and yaw angular acceleration. For linear acceleration and pitch-angular and roll-angular accelerations, Auto-Regressive Moving Average with exogenous inputs (ARMAX) models were identified. Yang in [34] developed an aerodynamic parameter estimation method and an adaptive attitude control method based on a Linear Active Disturbance Rejection

Controller (LADRC).

In [35], Lui used a closed-loop multi-variable extremum seeking algorithm for a nonlinear quadrotor helicopter parameter identification with two groups of input data. As step response experiments obtained the gradients of the performance parameters, the whole system searches along the negative gradient of the cost function until the reference trajectory or point is derived.

Cunningham in [36] stated that past research involving the system identification of a multirotor vehicle generally produced only a closed-loop model. This was because identifying the open-loop unstable dynamics of a multirotor is considered impractical. They used time-domain equation-error and frequency domain output-error techniques to estimate an open-loop model of a multirotor vehicle.

Wang in [37] presented a Lyapunov-MARI (Model Reference Adaptive Identification) algorithm, which only needs the UAV and some standard sensors to realize the parameter identification. Firstly, they determined specific parameters that are not easy to be measured in the quadrotor UAV model. These parameters mainly include the three-axis moments of inertia, the rotors lift coefficient, and drag coefficient. Then, they improved the traditional period measurement method of the Rope Suspension Approach for the moments of inertia. For the lift and the drag coefficients, the Lyapunov-MARI algorithm was proposed for parameter identification. The results showed that the flight state calculated by the identified parameters was compatible with the actual flight data.

Mohajerin in [4] proposed a hybrid model with two configurations predicting the system state over many future steps using only the input by

combining deep recurrent neural networks with a quadrotor motion model.

In [38], Jiang found the parameters of a mathematical model developed for the longitudinal motion of a small UAV using Particle Swarm Optimization (PSO). Brain in [39] introduced a process that automatically updated aircraft simulation models using flight data. They achieved this by constructing a model of a transport aircraft and used it to collect flight data. Then the data was used to estimate the parameters of the mathematically developed model for the body-axis normal force coefficient. [40] offered a method for system identification of a Small Unmanned Helicopter (SUH) system using Improved Particle Swarm Optimization (IPSO). The collected data was used to find the aerodynamic parameters of the state-space model developed. They verified the accuracy of the resulting model by comparing the time-domain response of the simulation model and the actual flight.

While it is clear that an accurate math model is needed to enhance the performance of virtual dynamics emulation, which eventually results in higher accuracy of the candidate controller, no one to date has developed a cohesive model for utilization in a testbed that enables the development of controllers and hardware. Given the recent attempts for identifying an accurate rotorcraft model by considering the aerodynamic effects, it stands to reason that considering these effects in design and control will enhance the performance, particularly when paired with modern artificial intelligence methods. As a result, a gray box model of the drone is developed in this research that uses the particle swarm optimization algorithm to fully identify the actual parameters of the drone's dynamics and aerodynamics.

1.3.3 Custom drone software using automatic code generation

Different works have been done to incorporate auto code generation into embedded programming workflow using various tools and for different situations, even for critical applications [41, 42, 43, 44, 45, 46]. Many NASA projects have used Mathworks Simulink, RealTime Workshop, and Embedded Coder to model code development and generate actual C flight code from Simulink models, particularly in the Guidance Navigation and Control domain. NASA's Orion spacecraft has extensively used code generators to develop flight software [47]. For the PROBA-2 spacecraft, model-based (MatrixX/SystemBuild) technologies were used for code generation of the onboard software [48]. In the "prototyping as C/C++ code" chapter of [49], Zarrinkoub talked about generating C/C++ code for algorithms of the latest mobile communications.

In my research, the feed-forward controller, Vicon Camera integration, the closed-loop controller, command handling, take-off logic, the landing controller, and the crash detector were developed in Matlab with the custom-written code. Then, automatic code generation was used to implement and deploy the algorithms on the hardware. As discussed in 1.4.3, this custom open software gives a high level of flexibility and access, making it possible to implement the custom algorithms needed for this research. Using the best practices discussed in Section §6.2, appropriate C/C++ code can be generated from Matlab code.

1.4 Contributions

1.4.1 Testbed for the development of new technologies

This thesis presents and develops a drone testing platform for easier development of new hardware and software algorithms. Using this testbed, it is possible to test and qualify new technologies in a hardware-in-the-loop environment, and so it reaches a technology readiness level of 6-7 (which needs testing in a relevant or space environment). This will eliminate the need for costly launches or flights, microgravity emulators, and most kinds of unique test environments. As a result, this would drastically decrease the cost and time spent on the qualification activities.

The practical use of new technologies in this testbed and their resultant higher readiness makes it possible to use them in industrial applications, which allows them to resolve the challenges mentioned above.

Figure 1.2 shows a conceptual diagram of the testbed.

The first part is called "Drone Dynamic Compensator," which eliminates on-earth drone dynamics and provides sufficient abstraction for independent drone control. This includes removing gravity (so the drone does not fall) and eliminating friction (so after an initial imposed impact force, the drone will continue to move with a constant velocity).

"Virtual Dynamic Engine" emulates different types of virtual environments, including different gravity levels, different air viscosities, Mars environment, etc. These virtual dynamics can be represented by special environment forces as feed-forward control augmentations (e.g., custom gravity or friction).

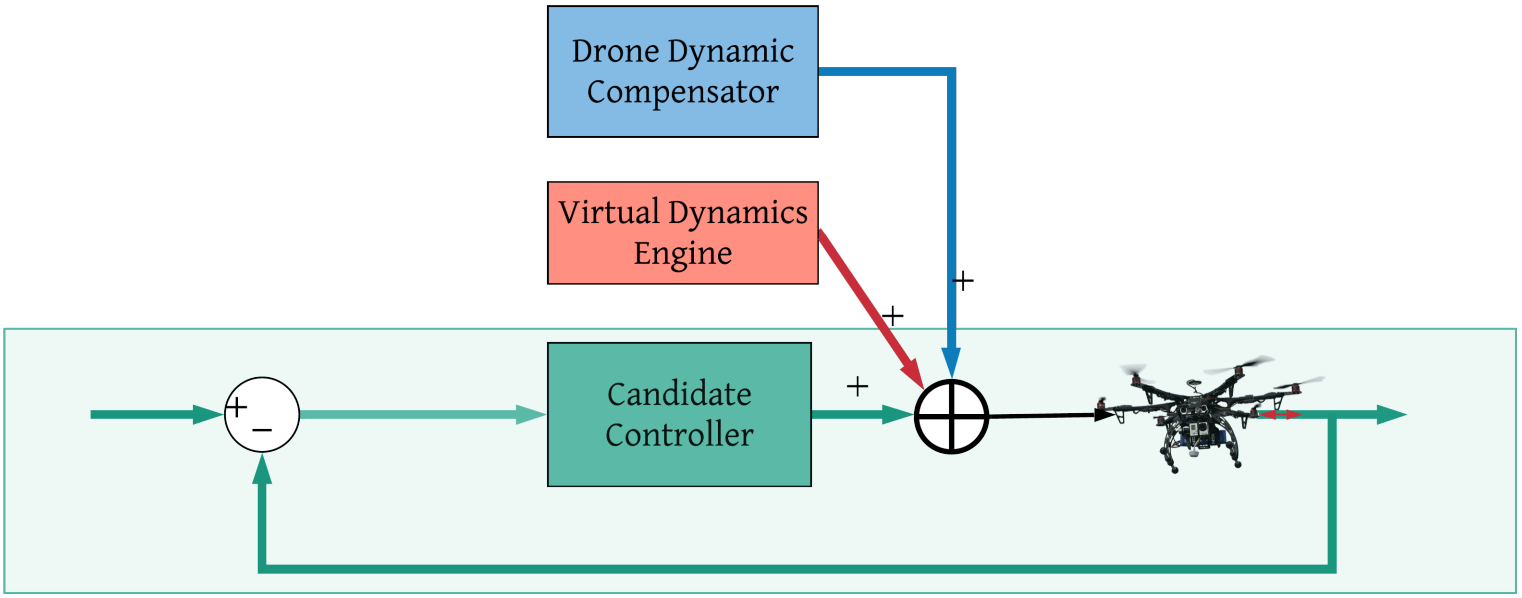

Figure 1.2: Drone testbed block diagram

Finally built on top of the created virtual environment, the "Candidate Controller" allows us to test any spacecraft or drone control algorithms. These candidate controllers can be the state of art control algorithms that have never been tested on a drone or in space.

1.4.2 System-identification of drones

To find an accurate model of the drone, this research develops a smart system identification procedure. This method utilizes flight data to achieve a high fidelity model using the PSO metaheuristic algorithm on a gray box model. Using this method, it is possible to easily construct the model of any drone without wind tunnel tests. The resultant mathematical model of the plant is programmable, which makes it practical to implement it on the embedded

processors.

In addition to virtual environment simulation, an accurate drone model helps in other applications. The increasing number of drones and their broad applications makes it crucial to design high-performance drone hardware. Additionally, performing sophisticated flights and operations such as high-speed maneuvers is impossible unless using high-fidelity flight controllers (e.g. model predictive control). Using a model acquired from this research makes it possible to improve the design of the drone hardware and the performance of the controllers.

1.4.3 Custom drone software using automatic code generation

In this research, the implementation of this testbed's flight and ground software is done by automatically generating the C/C++ code. This allows hardware and processor in-the-loop software development while performing simulations and debugging at the same time. Using the same code for hardware implementation and software simulation gives flexibility in terms of the algorithm used in this research. This is because I am no longer limited to off-the-shelf hardware with built-in software. Any algorithm can be quickly developed and tested in a simulation environment, and it can be conveniently deployed to real hardware. In this research, the software for the feed-forward controller, Vicon Camera integration, the closed-loop controller, command handling, take-off logic, the landing controller, and the crash detector were developed with custom-written code and then deployed to the hardware using auto code generation.

In addition to the onboard software, all of the operations of the system identification are programmed for a model that is developed inside Matlab. This Matlab model is also used to explore the implementation of new control algorithms (open and closed loop) for the drone.

1.5 Outline

In this research, by using the flight data, a system identification method is used. First, a mathematical model of drone dynamics and aerodynamics is developed in Chapter 2. Later in Chapter 5, this model is used in the gray box system identification. Particle swarm optimization is utilized for finding the parameters that make the mathematical model act like the collected data. A closed-loop controller is implemented for preparing the flight data (Chapter 3), which is also used for drone flight simulations and flight software evaluation. Finally, this research program evaluates the utility of these models to develop a feed-forward control method (Chapter 4) which is directly used for the emulation of virtual environments. In Chapter 6, the details of the software used in the implementation of the testbed are discussed. In this chapter, the methods used for auto-code generation are discussed. Finally, in Chapter 7, the results of the feed-forward controller are presented.

Chapter 2

Equations of the Motion

2.1 Kinematics

It is considered that an inertial frame (x, y, z) is attached to ground that describes the location of the drone (r^i) as[1]:

$$r^i = \begin{bmatrix} x \\ y \\ z \end{bmatrix} \tag{2.1}$$

A body-fixed frame (x^b, y^b, z^b) is attached to the drone, in which the z^b axis is perpendicular to the drone plane, and x^b for both + and × configurations is as shown in Figure 2.1. The center of the frame is located in the drone's center of mass.

The angles between arms and x^b-axis for × and + configuration of motors

[1]In this thesis, the superscript of a vector (e.g. such as i in r^i) shows the frame in which that vector is described in.

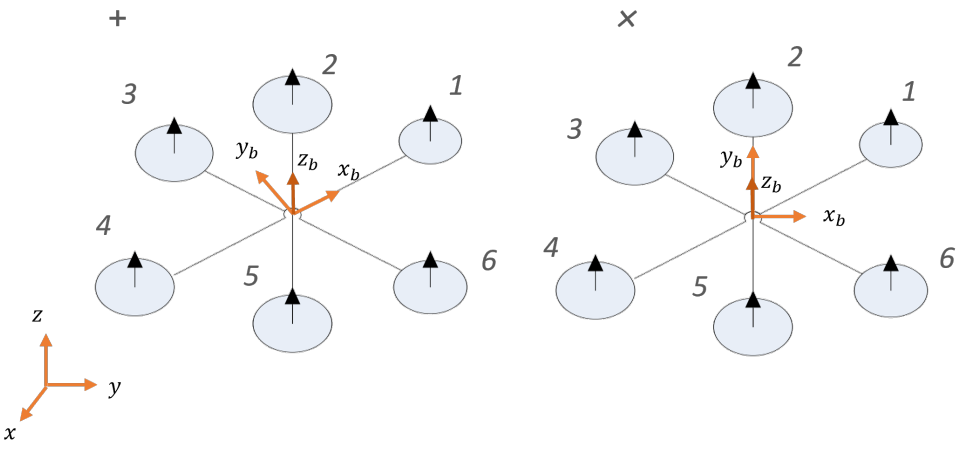

Figure 2.1: Drone frame set and motor configuration

are found from Equation (2.2)

$$a_n = \begin{cases} -\frac{2\pi}{N_{motor}}(n-1) & + \text{ configuration} \\ \frac{2\pi}{N_{motor}}(n-1) + \frac{\pi}{N_{motor}} & \times \text{ configuration} \end{cases} \tag{2.2}$$

where a_n is the angle of each arm with respect to x axis, n is the arm number, and N_{motor} is the total number of motors. If a quadrotor with 4 arms is used, $N_{motor} = 4$.

In the $\times$ configuration, a_n becomes

$$a_n = \frac{2\pi}{4}(n-1) + \frac{\pi}{4}, n = 1, 2, 3, 4$$

The above is simplified to

$$a_1 = \frac{\pi}{4},\ a_2 = \frac{3\pi}{4},\ a_3 = \frac{5\pi}{4},\ a_4 = \frac{7\pi}{4} \tag{2.3}$$

In the + configuration, a_n becomes

$$a_n = -\frac{2\pi}{4}\left(n-1\right), n = 1, 2, 3, 4$$

The above is calculated as:

$$a_1 = 0,\ a_2 = \frac{\pi}{2},\ a_3 = \frac{3\pi}{2},\ a_4 = \frac{5\pi}{2} \tag{2.4}$$

The orientation of the drone is described in terms of Euler angles. The order of the rotation is considered to be a sequential rotation about $x, y',$ and z'' (intermediate body-fixed axes). So yaw θ_3, pitch θ_2, and roll θ_1 describe the attitude of the drone as shown:

$$\text{Euler angles: } \theta = \begin{bmatrix} \theta_1 \\ \theta_2 \\ \theta_3 \end{bmatrix} \tag{2.5}$$

$$(i \text{ frame})\ xyz \xrightarrow{\theta_1 \text{ about } x} x'y'z' \xrightarrow{\theta_2 \text{ about } y'} x''y''z'' \xrightarrow{\theta_3 \text{ about } z''} x^by^bz^b\ (b \text{ frame}) \tag{2.6}$$

If a vector is described in the body-fixed frame (v^b), to describe it back in the inertial frame, the rotations about $x, y',$ and z'' should be reversed until

the inertial description (v^i) is found.

$$v^i = R_{i\leftarrow b} v^b$$

$$R_{i\leftarrow b} = R_{i\leftarrow'}(\theta_1) R_{'\leftarrow''}(\theta_2) R_{''\leftarrow b}(\theta_3) \tag{2.7}$$

The rotation matrices in Equation (2.7) that reverse the rotations about $z'', y',$ and x are respectively given as[2]:

$$R_{''\leftarrow b}(\theta_3) = \begin{bmatrix} c_3 & s_3 & 0 \\ -s_3 & c_3 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{2.8}$$

$$R_{'\leftarrow''}(\theta_2) = \begin{bmatrix} c_2 & 0 & -s_2 \\ 0 & 1 & 0 \\ s_2 & 0 & c_2 \end{bmatrix} \tag{2.9}$$

$$R_{i\leftarrow'}(\theta_1) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & c_1 & s_1 \\ 0 & -s_1 & c_1 \end{bmatrix} \tag{2.10}$$

By multiplying these matrices as Equation (2.7), the rotation matrix for this sequence becomes:

[2]For simplicity, trigonometric functions (such as $cos(\theta_1)$ and $sin(\theta_1)$) are notated in their short form (e.g. c_1 and s_1)

$$R_{i\leftarrow b} = R = \begin{bmatrix} c_2c_3 & c_2s_3 & -s_2 \\ s_1s_2c_3 - c_1s_3 & s_1s_2s_3 + c_1c_3 & s_1c_2 \\ c_1s_2c_3 + s_1s_3 & c_1s_2s_3 - s_1c_3 & c_1c_2 \end{bmatrix} \tag{2.11}$$

For brevity, $R_{i\leftarrow b}$ will be notated just as R.

By taking the time derivative of the rotation matrix, the following matrix is calculated.

$$\dot{R} = \begin{bmatrix} -c_3s_2\dot{\theta}_2 - c_2s_3\dot{\theta}_3 \\ (s_1s_3 + c_1s_2c_3)\dot{\theta}_1 + s_1c_2c_3\dot{\theta}_2 + (-c_1c_3 - s_1s_2s_3)\dot{\theta}_3 \\ (c_1s_3 - s_1s_2c_3)\dot{\theta}_1 + c_1c_2c_3\dot{\theta}_2 + (s_1c_3 - c_1s_2s_3)\dot{\theta}_3 \\ -s_2s_3\dot{\theta}_2 + c_2c_3\dot{\theta}_3 \\ (-s_1c_3 + c_1s_2s_3)\dot{\theta}_1 + s_1c_2s_3\dot{\theta}_2 + (-c_1s_3 + s_1s_2c_3)\dot{\theta}_3 \\ (-c_1c_3 - s_1s_2s_3)\dot{\theta}_1 + c_1c_2s_3\dot{\theta}_2 + (s_1s_3 + c_1s_2c_3)\dot{\theta}_3 \\ -c_2\dot{\theta}_2 \\ c_1c_2\dot{\theta}_1 - s_1s_2\dot{\theta}_2 \\ -s_1c_2\dot{\theta}_1 - c_1s_2\dot{\theta}_2 \end{bmatrix} \tag{2.12}$$

The angular velocity of the drone with respect to the inertial frame and expressed in the body-fixed frame ($\omega^{b\times}$) is [50]:

$$\omega^{b\times} = R^T\dot{R} \tag{2.13}$$

Substituting equation (2.11) and Equations (2.12) gives:

$$\omega^{b\times} = R^T\dot{R} = \begin{pmatrix} 0 & -s_2\dot{\theta}_1 + \dot{\theta}_3 & -c_2s_3\dot{\theta}_1 - c_3\dot{\theta}_2 \\ s_2\dot{\theta}_1 - \dot{\theta}_3 & 0 & c_2c_3\dot{\theta}_1 - s_3\dot{\theta}_2 \\ c_2s_3\dot{\theta}_1 + c_3\dot{\theta}_2 & -c_2c_3\dot{\theta}_1 + s_3\dot{\theta}_2 & 0 \end{pmatrix} \tag{2.14}$$

The angular velocity vector ($\omega^b = \begin{bmatrix} \omega_1 \\ \omega_2 \\ \omega_3 \end{bmatrix}$) is then found by un-skew-symmetric of the $\omega^{b\times}$ matrix:

$$\omega^{b\times} = R^T\dot{R} \xrightarrow{\text{un-skew-symmetric}(\omega^{b\times})} \omega^b = \begin{bmatrix} \omega_1 \\ \omega_2 \\ \omega_3 \end{bmatrix} = \begin{pmatrix} -c_2c_3\dot{\theta}_1 + s_3\dot{\theta}_2 \\ -c_2s_3\dot{\theta}_1 - c_3\dot{\theta}_2 \\ s_2\dot{\theta}_1 - \dot{\theta}_3 \end{pmatrix} \tag{2.15}$$

From Equations (2.15), by separating the rate of the Euler angles from the angular velocity vector, equation (2.16) is achieved, in which the matrix Υ relates the time derivative of the Euler angles to the angular velocity vector.

$$\underbrace{\begin{bmatrix} \omega_1 \\ \omega_2 \\ \omega_3 \end{bmatrix}}_{\omega^b} = \underbrace{\begin{pmatrix} -c_2c_3 & s_3 & 0 \\ -c_2s_3 & -c_3 & 0 \\ s_2 & 0 & -1 \end{pmatrix}}_{\Upsilon} \begin{bmatrix} \dot{\theta}_1 \\ \dot{\theta}_2 \\ \dot{\theta}_3 \end{bmatrix} \tag{2.16}$$

If Υ is invertible in the angle space that drone can fly, the derivative of Euler angles is found by pre-multiplying both sides of Equations (2.16) by Υ^{-1}:

$$\begin{bmatrix} \dot{\theta}_3 \\ \dot{\theta}_2 \\ \dot{\theta}_1 \end{bmatrix} = \underbrace{\begin{pmatrix} -\frac{c_3}{c_2} & -\frac{s_3}{c_2} & 0 \\ s_3 & -c_3 & 0 \\ -c_3 \tan\theta_2 & -s_3 \tan\theta_2 & -1 \end{pmatrix}}_{\Upsilon^{-1}} \underbrace{\begin{bmatrix} \omega_1 \\ \omega_2 \\ \omega_3 \end{bmatrix}}_{\omega^b} \tag{2.17}$$

2.2 Dynamics

The states that describe the motion of a drone are x, y, z, the position, $\theta_1, \theta_2, \theta_3$, the attitude, $\dot{x}, \dot{y}, \dot{z}$, the velocity, and the angular velocity $\omega_1, \omega_2, \omega_3$. The state vector and the derivative of the state vector are shown in Equation (2.18).

$$\begin{aligned} s &= \begin{bmatrix} x & y & z & \theta_1 & \theta_2 & \theta_3 & \dot{x} & \dot{y} & \dot{z} & \omega_1 & \omega_2 & \omega_3 \end{bmatrix}^T \\ \dot{s} &= \begin{bmatrix} \dot{x} & \dot{y} & \dot{z} & \dot{\theta}_1 & \dot{\theta}_2 & \dot{\theta}_3 & \ddot{x} & \ddot{y} & \ddot{z} & \dot{\omega}_1 & \dot{\omega}_2 & \dot{\omega}_3 \end{bmatrix}^T \end{aligned} \tag{2.18}$$

To form the state space equations, I need to rewrite the derivative of states ($\dot{s}$) in terms of states (s) and external forces/moments.

The first three elements of $\dot{s}$ are already the states:

$$\begin{bmatrix} \dot{s_1} \\ \dot{s_2} \\ \dot{s_3} \end{bmatrix} = \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{z} \end{bmatrix} = \begin{bmatrix} s_7 \\ s_8 \\ s_9 \end{bmatrix} \tag{2.19}$$

The next three elements of $\dot{s}$ can be calculated from Equation (2.17):

$$\begin{bmatrix} \dot{s_4} \\ \dot{s_5} \\ \dot{s_6} \end{bmatrix} = \begin{bmatrix} \dot{\theta}_3 \\ \dot{\theta}_2 \\ \dot{\theta}_1 \end{bmatrix} = \begin{pmatrix} -\frac{c_3}{c_2} & -\frac{s_3}{c_2} & 0 \\ s_3 & -c_3 & 0 \\ -c_3 \tan\theta_2 & -s_3 \tan\theta_2 & -1 \end{pmatrix} \begin{bmatrix} \omega_1 \\ \omega_2 \\ \omega_3 \end{bmatrix} \tag{2.20}$$

To complete the state space equations, Newton-Euler equations of the motion are used as follows:

$$m\ddot{r}^i = F^i_{\text{tot}} \qquad \text{in inertial frame} \tag{2.21}$$

$$I\dot{\omega}^b + \omega^b \times I\omega^b = M^b_{\text{tot}} \qquad \text{in body-fixed frame} \tag{2.22}$$

In Equation (2.21), r^i is the position vector, m is the mass, F^i_{tot} is the total external forces on the drone in the inertial frame, and in Equation (2.22), I is the moment inertia matrix in the body-fixed frame, and M^b_{tot} is the total external moments on the drone in the body-fixed frame about the drone's center of mass.

By rearranging Equation (2.21) and equation (2.22), the following is obtained:

$$\ddot{r}^i = \frac{1}{m} F^i_{\text{tot}} \tag{2.23}$$

$$\dot{\omega}^b = I^{-1}\left(M^b_{\text{tot}} - \omega^b \times I\omega^b\right) \tag{2.24}$$

So, the the last six elements of $\dot{s}$ are calculated from Equation (2.23) and Equation (2.24).

$$\begin{bmatrix} \dot{s_7} \\ \dot{s_8} \\ \dot{s_9} \end{bmatrix} = \begin{bmatrix} \ddot{x} \\ \ddot{y} \\ \ddot{z} \end{bmatrix} = \ddot{r}^i = \frac{1}{m} F^i_{\text{tot}} \tag{2.25}$$

$$\begin{bmatrix} \dot{s_{10}} \\ \dot{s_{11}} \\ \dot{s_{12}} \end{bmatrix} = \begin{bmatrix} \dot{\omega_1} \\ \dot{\omega_2} \\ \dot{\omega_3} \end{bmatrix} = \dot{\omega}^b = I^{-1} \left(M^b_{\text{tot}} - \omega^b \times I\omega^b \right) \tag{2.26}$$

Putting Equation (2.19), Equation (2.20), Equation (2.25), and Equation (2.26) all together, the full state space equation is calculated as:

$$\dot{r}^i = \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{z} \end{bmatrix} = \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{z} \end{bmatrix}$$

$$\dot{\theta} = \begin{bmatrix} \dot{\theta}_3 \\ \dot{\theta}_2 \\ \dot{\theta}_1 \end{bmatrix} = \begin{pmatrix} -\frac{c_3}{c_2} & -\frac{s_3}{c_2} & 0 \\ s_3 & -c_3 & 0 \\ -c_3 tan\theta_2 & -s_3 tan\theta_2 & -1 \end{pmatrix} \begin{bmatrix} \omega_1 \\ \omega_2 \\ \omega_3 \end{bmatrix}$$

$$\ddot{r}^i = \begin{bmatrix} \ddot{x} \\ \ddot{y} \\ \ddot{z} \end{bmatrix} = \frac{1}{m} F^i_{\text{tot}}$$

$$\dot{\omega}^b = \begin{bmatrix} \dot{\omega_1} \\ \dot{\omega_2} \\ \dot{\omega_3} \end{bmatrix} = I^{-1} \left(M^b_{\text{tot}} - \omega^b \times I\omega^b \right) \tag{2.27}$$

Later in 5.2.2, Equation (2.27) is numerically integrated to calculate the states of the drone over time.

2.3 Aerodynamics

2.3.1 Aerodynamics model

The gravity force on the drone is

$$F^i_{gravity} = -mg\hat{k} = -mg \begin{bmatrix} 0 \\ 0 \\ -1 \end{bmatrix} \tag{2.28}$$

Where m is the drone mass, g is the acceleration due to gravity on Earth, and $\hat{k}$ is the direction vector in the z direction.

The total force F^i_{tot} applied to the drone consists of the aerodynamic force F^b_{aero} and the gravity force $F^i_{gravity}$. The aerodynamic forces (F^b_{aero}) are described in the body fixed-frame. So the total force applied to the drone in the inertial frame (F^i_{tot}) is found by pre-multiplying rotation matrix R at F^b_{aero} and substituting Equation (2.28):

$$F^i_{tot} = RF^b_{aero} - mg\hat{k} \tag{2.29}$$

The total moment M^b_{tot} to the drone is only caused by the aerodynamic moments around the drone's center of mass. Since the total moment and aerodynamics moment are described in the body-fixed frame, the following is obtained.

$$M^b_{\text{tot}} = M^b_{\text{aero}} \tag{2.30}$$

To model the aerodynamic forces F^b_{aero} and moments M^b_{aero}, a model including the thrust force and the drag moment is considered [51].

The thrust force (F^b_T) is the vertical force acting on a propeller. Thrust force is a square function of the motor's rotation speed (ω_{motor}). For one propeller, this force is calculated from Equation (2.31), in which C_{F_T} constant is the thrust force coefficient.

$$F^b_T = \rho A_{prop} R^2_{prop}\ C_{F_T} \omega^2_{motor} \qquad \text{for each propeller} \tag{2.31}$$

Where ρ is the air density, A_{prop} is the area that propellers take when they rotate, and R_{prop} is the propeller radius.

The drag moment (M_D) is the moment about the rotor shaft caused by the aerodynamic drag on a propeller. The drag moment is a square function of the motor's rotation speed. For one propeller, this moment is calculated from 2.32, in which C_{M_D} is the drag moment coefficient.

$$M^b_D = \text{sign}(\omega_{motor})\ \rho A_{prop} R^3_{prop}\ C_{M_D} \omega^2_{motor} \qquad \text{for each propeller} \tag{2.32}$$

The direction of the motor rotation specifies sign(ω_{motor}).

In Equation (2.31) and Equation (2.32), the area that the propellers make when they rotate (A_{prop}) can be calculated as:

$$A_{prop} = \pi R_{prop}^2 \tag{2.33}$$

The thrust forces also cause moments about the roll (x) and pitch (y) axes ($M_{F_T,x}, M_{F_T,y}$). These moments for the nth arm are calculated from Equation (2.34).

$$\begin{aligned} M_{F_T,x}^b &= F_T^b l \sin(a_n) \\ M_{F_T,y}^b &= -F_T^b l \cos(a_n) \end{aligned} \qquad \text{for each propeller} \tag{2.34}$$

Where l is the arm length.

By summing Equation (2.31), Equation (2.32), and Equation (2.34), the aerodynamic forces and moments for all of the motors are found as in Equation (2.35).

$$\begin{aligned} F_{aero,x}^b &= 0 \\ F_{aero,y}^b &= 0 \\ F_{aero,z}^b &= \sum_{n=1}^{N_{motor}} F_{T_n}^b \\ M_{aero,x}^b &= \sum_{n=1}^{N_{motor}} F_{T_n}^b l \sin(a_n) \\ M_{aero,y}^b &= \sum_{n=1}^{N_{motor}} -F_{T_n}^b l \cos(a_n) \\ M_{aero,z}^b &= \sum_{n=1}^{N_{motor}} M_{D_n}^b \end{aligned} \qquad n = 1, 2, ..., N_{motor} \tag{2.35}$$

where N_{motor} is the total number of motors.

Putting Equation (2.35) in the vector format gives:

$$F^b_{aero} = \begin{bmatrix} 0 \\ 0 \\ \sum_{n=1}^{N_{motor}} F^b_{T_n} \end{bmatrix} \tag{2.36}$$

$$M^b_{aero} = \begin{bmatrix} \sum_{n=1}^{N_{motor}} F^b_{T_n} lsin\left(a_n\right) \\ \sum_{n=1}^{N_{motor}} -F^b_{T_n} lcos\left(a_n\right) \\ \sum_{n=1}^{N_{motor}} M^b_{D_n} \end{bmatrix} \tag{2.37}$$

The substitution of Equation (2.31) and Equation (2.32) into the above equations will give Equation (2.38) and Equation (2.39):

$$F^b_{aero} = \begin{bmatrix} 0 \\ 0 \\ \sum_{n=1}^{N_{motor}} \rho A_{prop} R^2_{prop} C_{F_{T_n}} \omega^2_{motor_n} \end{bmatrix} \tag{2.38}$$

$$M^b_{aero} = \begin{bmatrix} \sum_{n=1}^{N_{motor}} \rho A_{prop} R^2_{prop} C_{F_{T_n}} \omega^2_{motor_n} lsin\left(a_n\right) \\ \sum_{n=1}^{N_{motor}} -\rho A_{prop} R^2_{prop} C_{F_{T_n}} \omega^2_{motor_n} lcos\left(a_n\right) \\ \sum_{n=1}^{N_{motor}} \text{sign}(\omega_{motor})\ \rho A_{prop} R^3_{prop}\ C_{M_{D_n}}\ \omega^2_{motor_n} \end{bmatrix} \tag{2.39}$$

By assuming that ρ, A_{prop}, R and l are the same for all rotors, Equation (2.38) and Equation (2.39) are simplified to:

$$F^b_{aero} = \rho A_{prop} R^2_{prop} \begin{bmatrix} 0 \\ 0 \\ \sum_{n=1}^{N_{motor}} C_{F_{T_n}} \omega^2_{motor_n} \end{bmatrix} \tag{2.40}$$

$$M^b_{aero} = \rho A_{prop} R^2_{prop} \begin{bmatrix} l \sum_{n=1}^{N_{motor}} C_{F_{T_n}} \omega^2_{motor_n} sin\,(a_n) \\ -l \sum_{n=1}^{N_{motor}} C_{F_{T_n}} \omega^2_{motor_n} cos\,(a_n) \\ R_{prop} \sum_{n=1}^{N_{motor}} \text{sign}(\omega_{motor}) C_{M_{D_n}}\ \omega^2_{motor_n} \end{bmatrix} \tag{2.41}$$

2.4 State-space equations

By substitution of Equation (2.40) into Equation (2.29) ($F^i_{tot} = RF^b_{aero} - mg\hat{k}$) and Equation (2.30) ($M^b_{\text{tot}} = M^b_{\text{aero}}$), the total forces and moments applied to the drone are obtained as:

$$F^i_{tot} = R\ \rho A_{prop} R^2_{prop} \begin{bmatrix} 0 \\ 0 \\ \sum_{n=1}^{N_{motor}} C_{F_{T_n}} \omega^2_{motor_n} \end{bmatrix} - \begin{bmatrix} 0 \\ 0 \\ mg \end{bmatrix} \tag{2.42}$$

$$M^b_{tot} = \rho A_{prop} R^2_{prop} \begin{bmatrix} l \sum_{n=1}^{N_{motor}} C_{F_{T_n}} \omega^2_{motor_n} sin\,(a_n) \\ -l \sum_{n=1}^{N_{motor}} C_{F_{T_n}} \omega^2_{motor_n} cos\,(a_n) \\ R_{prop} \sum_{n=1}^{N_{motor}} \text{sign}(\omega_{motor}) C_{M_{D_n}}\ \omega^2_{motor_n} \end{bmatrix} \tag{2.43}$$

By plugging these into Equation (2.23) ($\ddot{r}^i = \frac{1}{m} F^i_{\text{tot}}$) and Equation (2.24) ($\dot{\omega}^b = I^{-1}\left(M^b_{\text{tot}} - \omega^b \times I\omega^b\right)$), $\ddot{r}^i$ and $\dot{\omega}^b$ are expanded to:

$$\ddot{r}^i = R\ \frac{\rho A_{prop} R^2_{prop}}{m} \begin{bmatrix} 0 \\ 0 \\ \sum_{n=1}^{N_{motor}} C_{F_{T_n}} \omega^2_{motor_n} \end{bmatrix} - \begin{bmatrix} 0 \\ 0 \\ g \end{bmatrix} \tag{2.44}$$

$$\dot{\omega}^b = I^{-1} \left(\rho A_{prop} R^2_{prop} \begin{bmatrix} l \sum_{n=1}^{N_{motor}} C_{F_{T_n}} \omega^2_{motor_n} sin\left(a_n\right) \\ -l \sum_{n=1}^{N_{motor}} C_{F_{T_n}} \omega^2_{motor_n} cos\left(a_n\right) \\ R_{prop} \sum_{n=1}^{N_{motor}} \text{sign}(\omega_{motor}) C_{M_{D_n}}\ \omega^2_{motor_n} \end{bmatrix} - \omega^b \times I\omega^b \right) \tag{2.45}$$

$\ddot{r}^i$ and $\dot{\omega}^b$ are used on the right-hand side of the last six rows of the state space equations (2.27), which is numerically integrated to calculate the states of the drone over time in Section 5.2.2.

2.5 Quadrotor aerodynamics

If the dynamics of a quadrotor is considered, the number of motors is four ($N_{motor} = 4$), and so the vector of motor speeds (ω_{motor}) is defined as

$$\omega_{motor} = \begin{bmatrix} \omega_{motor,1} \\ \omega_{motor,2} \\ \omega_{motor,3} \\ \omega_{motor,4} \end{bmatrix} \tag{2.46}$$

By factoring out the square of motor speeds (Equation (2.46)) from Equation (2.40) and only considering the third non-zero row ($F^b_{aero,z}$), the following can be written:

$$F^b_{aero,z} = \rho A_{prop} R^2_{prop} \begin{bmatrix} C_{F_{T_1}} & C_{F_{T_2}} & C_{F_{T_3}} & C_{F_{T_4}} \end{bmatrix} \begin{bmatrix} \omega^2_{motor,1} \\ \omega^2_{motor,2} \\ \omega^2_{motor,3} \\ \omega^2_{motor,4} \end{bmatrix} \tag{2.47}$$

By expanding Equation (2.41) for 4 motors and factoring out the square of motor speeds (Equation (2.46)), the following can be written:

$$M^b_{aero} = \rho A_{prop} R^2_{prop} \begin{bmatrix} lC_{F_{T_1}} sin(a_1) & lC_{F_{T_2}} sin(a_2) & lC_{F_{T_3}} sin(a_3) & lC_{F_{T_4}} sin(a_4) \\ -lC_{F_{T_1}} cos(a_1) & -lC_{F_{T_2}} cos(a_2) & -lC_{F_{T_3}} cos(a_3) & -lC_{F_{T_4}} cos(a_4) \\ R_{prop} C_{M_{D_1}} & -R_{prop} C_{M_{D_2}} & R_{prop} C_{M_{D_3}} & -R_{prop} C_{M_{D_4}} \end{bmatrix} \begin{bmatrix} \omega^2_{motor,1} \\ \omega^2_{motor,2} \\ \omega^2_{motor,3} \\ \omega^2_{motor,4} \end{bmatrix} \tag{2.48}$$

Putting Equation (2.47) and Equation (2.48) together gives the following equation:

$$\begin{bmatrix} F^b_{aero,z} \\ M^b_{aero} \end{bmatrix} = \rho A_{prop} R^2_{prop} \begin{bmatrix} C_{F_{T_1}} & C_{F_{T_2}} & C_{F_{T_3}} & C_{F_{T_4}} \\ lC_{F_{T_1}} sin(a_1) & lC_{F_{T_2}} sin(a_2) & lC_{F_{T_3}} sin(a_3) & lC_{F_{T_4}} sin(a_4) \\ -lC_{F_{T_1}} cos(a_1) & -lC_{F_{T_2}} cos(a_2) & -lC_{F_{T_3}} cos(a_3) & -lC_{F_{T_4}} cos(a_4) \\ R_{prop} C_{M_{D_1}} & -R_{prop} C_{M_{D_2}} & R_{prop} C_{M_{D_3}} & -R_{prop} C_{M_{D_4}} \end{bmatrix} \begin{bmatrix} \omega^2_{motor,1} \\ \omega^2_{motor,2} \\ \omega^2_{motor,3} \\ \omega^2_{motor,4} \end{bmatrix} \tag{2.49}$$

Later in Section §3.2, Equation (2.49) is used for the closed-loop control of the quadrotor.

Chapter 3

Closed-loop Control

To collect proper data for the identification of the drone model, it is needed to implement software that autonomously performs the flights. For doing so, a closed-loop controller is necessary. In this chapter, the methods and the mathematics behind the closed-loop controller are discussed.

As seen in Equation (2.44) and Equation (2.45), the control signals to this system are motor speeds, and the outputs of the system are its position and attitude. This is shown in Figure 3.1.

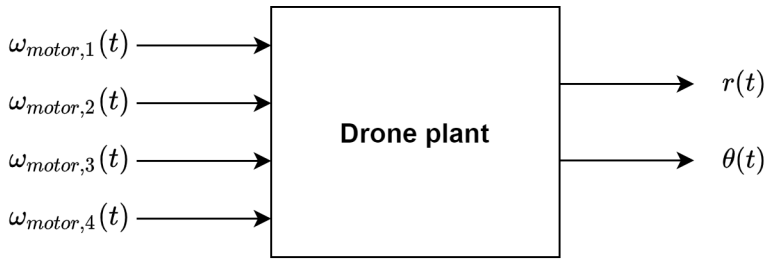

Figure 3.1: Drone plant

In Section §3.1, the control problem is defined, and in Section §3.2, the

controller commands are discussed. In Section §3.3, the detailed implementation of the controllers are discussed.

3.1 Control problem definition

The problem of the drone's closed-loop control is formulated as a path following problem. Consider that the drone needs to follow the known trajectory $r_{des}(t)$ with the desired yaw angle $\theta_{3,des}(t)$.

$$\text{The desired trajectory: } r_{des}(t),\ \theta_{3,des}(t)$$

where $r_{des}(t)$ is in the inertial frame, and $\theta_{3,des}(t)$ is the first Euler angle rotation (as discussed in Section §2.1). The block diagram of the whole system is shown in Figure 3.2.

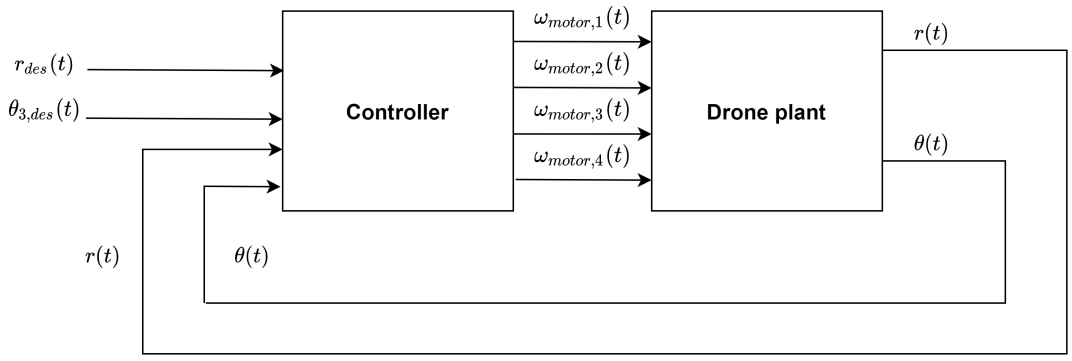

Figure 3.2: System block diagram. In this figure, the drone plant includes the sensors and the onboard estimator, and $\theta(t)$ and $r(t)$ are the output of the onboard estimator.

To form the control problem, the control error is defined as the difference

between the current position/angle and the desired position/angle

$$\begin{cases} e_r(t) = r_{des}(t) - r(t) \\ e_{\theta_3}(t) = \theta_{3,des}(t) - \theta_3(t) \end{cases} \tag{3.1}$$

Because the desired position and yaw angles are known at all times, and their curves are smooth, their time derivatives can be calculated. Taking the time derivative, the control error in Equation (3.1) is expanded into:

$$\begin{cases} e_r(t) = r_{des}(t) - r(t) \\ e_{\theta_3}(t) = \theta_{3,des}(t) - \theta_3(t) \end{cases} \rightarrow \begin{array}{c} \dot{e}_r(t) = \dot{r}_{des}(t) - \dot{r}(t) \\ \dot{e}_{\theta_3}(t) = \dot{\theta}_{3,des}(t) - \dot{\theta}_3(t) \end{array} \rightarrow \begin{array}{c} \ddot{e}_r(t) = \ddot{r}_{des}(t) - \ddot{r}(t) \\ \ddot{e}_{\theta_3}(t) = \ddot{\theta}_{3,des}(t) - \ddot{\theta}_3(t) \end{array} \tag{3.2}$$

In practice, the sensors measure the accelerations, angular velocities, and the position, which with the help of the onboard estimator, make r, $\dot{r}$, $\ddot{r}$,θ_3,$\dot{\theta}_3$, and $\ddot{\theta}_3$ known. This means that the position and yaw errors can be calculated from Equation (3.2) by the controller.

In the next section, the control commands of the drone are explained.

3.2 Control Commands

A general rigid body has 6 degrees of freedom of general motion as discussed in equation (2.18):

- x: translation in the x direction
- y: translation in the y direction
- z: translation in the z direction

- θ_3: yaw rotation
- θ_2: pitch rotation
- θ_1: roll rotation

However, the drone used in this research has a particular configuration of motors that does not give us fully decoupled degrees of control. So the system is considered an underactuated system. If it is assumed that the thrust force is always perpendicular to the drone surface, the control commands for this configuration and their relation to motor speeds is found as follows:

- Up and down translation (cased by F_T^i): by creating the equal thrusts in all the motors, I can translate the drone in the z direction (Figure 3.3).

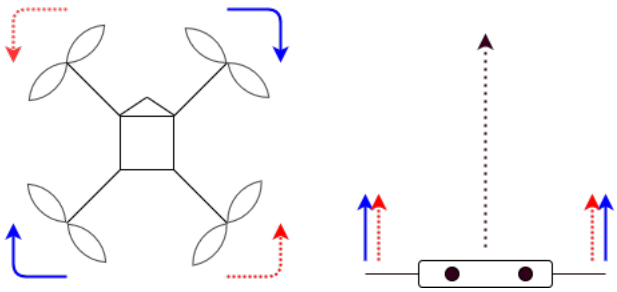

Figure 3.3: Translation of the drone in the z direction. The arrows in the left picture show the rotation of the propellers. In the right image, the arrows on the sides of the drone indicate the thrust forces of the propellers. The arrow in the middle of the drone shows the overall thrust force applied to the drone.

- Yaw rotation (caused by M_{yaw}^b): by rotating motors with the same speed in the diagonal motors, but with different rates for each diagonal, the

overall drag moment results in the yaw moment (M^b_{yaw}), which rotates the drone about the z^b axis (Figure 3.4).

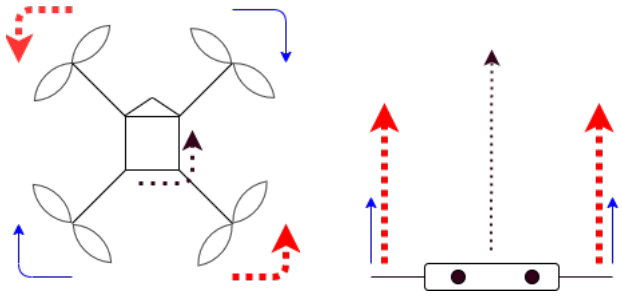

Figure 3.4: Yaw rotation of the drone. In the left picture, the arrows on the sides of the drone show the rotation of the propellers. The arrow in the middle shows the overall yaw moment. In the right image, the arrows on the sides of the drone indicate the thrust forces of the propellers. The arrow in the middle of the drone shows the overall thrust force applied to the drone.

- Roll rotation (caused byM^b_{roll}): by creating equal thrust on the sides of the drone, but with different values for each side, the differential thrusts on the sides result in an overall roll moment (M^b_{roll}), which rotates the drone about the x^b axis (Figure 3.5).

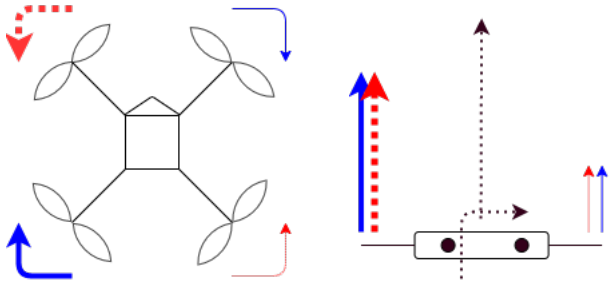

Figure 3.5: Roll rotation of the drone.In the left picture, the arrows on the sides of the drone show the rotation of the propellers. In the right image, the arrows on the sides of the drone indicate the thrust forces of the propellers, the vertical arrow in the middle of the drone shows the overall thrust force applied to the drone, and the last arrow shows the roll moment.

- Pitch rotation (caused byM^b_{pitch}): by creating equal thrust on the front and rear of the drone, but with different values for the front and back, the differential thrusts in the rear and front result in an overall pitch moment (M^b_{pitch}), which rotates the drone about the y^b axis (Figure 3.6).

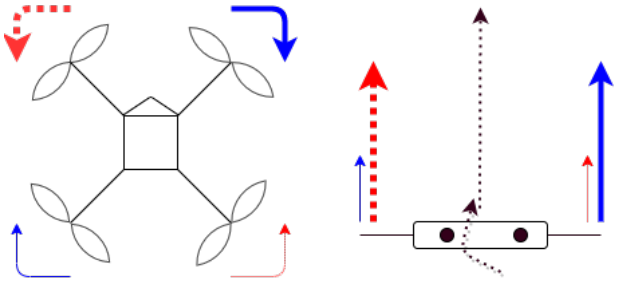

Figure 3.6: Pitch rotation of the drone. In the left picture, the arrows on the sides of the drone show the rotation of the propellers. In the right image, the arrows on the sides of the drone indicate the thrust forces of the propellers, the vertical arrow in the middle of the drone shows the overall thrust force applied to the drone, and the last arrow shows the pitch moment.

To formulate the above intuitive explanation, Equation (2.49) is expanded. This results in

$$\begin{bmatrix} F^b_{aero,z} \\ M^b_{aero} \end{bmatrix} = \begin{bmatrix} F^i_T \\ M^b_{yaw} \\ M^b_{pitch} \\ M^b_{roll} \end{bmatrix} \rightarrow$$

$$\begin{bmatrix} F^i_T \\ M_{yaw} \\ M_{pitch} \\ M^b_{roll} \end{bmatrix} = \rho A_{prop} R^2_{prop} \begin{bmatrix} C_{F_{T_1}} & C_{F_{T_2}} & C_{F_{T_3}} & C_{F_{T_4}} \\ lC_{F_{T_1}} sin(a_1) & lC_{F_{T_2}} sin(a_2) & lC_{F_{T_3}} sin(a_3) & lC_{F_{T_4}} sin(a_4) \\ -lC_{F_{T_1}} cos(a_1) & -lC_{F_{T_2}} cos(a_2) & -lC_{F_{T_3}} cos(a_3) & -lC_{F_{T_4}} cos(a_4) \\ R_{prop} C_{M_{D_1}} & -R_{prop} C_{M_{D_2}} & R_{prop} C_{M_{D_3}} & -R_{prop} C_{M_{D_4}} \end{bmatrix} \underbrace{\begin{bmatrix} \omega^2_{motor,1} \\ \omega^2_{motor,2} \\ \omega^2_{motor,3} \\ \omega^2_{motor,4} \end{bmatrix}}_{\omega^2_{motor}} \tag{3.3}$$

Equation (3.3) shows that the drone's controllers can adjust the thrust force and moments applied on the drone by changing the motor speeds (in a coupled way). So, the motor speeds are considered the control signals.

By inverting Equation (3.3)[1], a relation for calculating the motor speeds in terms of the aerodynamic forces and moments can be found, which is parametrically shown using m_{ij} in Equation (3.5).

[1]The design of the drone constants ensures that the right-hand side matrix is always invertible, and its determinant never becomes zero.

$$\underbrace{\begin{bmatrix} \omega^2_{motor,1} \\ \omega^2_{motor,2} \\ \omega^2_{motor,3} \\ \omega^2_{motor,4} \end{bmatrix}}_{\omega^2_{motor}} = \frac{1}{\rho A_{prop} R^2_{prop}} \begin{bmatrix} C_{F_{T_1}} & C_{F_{T_2}} & C_{F_{T_3}} & C_{F_{T_4}} \\ lC_{F_{T_1}} sin(a_1) & lC_{F_{T_2}} sin(a_2) & lC_{F_{T_3}} sin(a_3) & lC_{F_{T_4}} sin(a_4) \\ -lC_{F_{T_1}} cos(a_1) & -lC_{F_{T_2}} cos(a_2) & -lC_{F_{T_3}} cos(a_3) & -lC_{F_{T_4}} cos(a_4) \\ R_{prop} C_{M_{D_1}} & -R_{prop} C_{M_{D_2}} & R_{prop} C_{M_{D_3}} & -R_{prop} C_{M_{D_4}} \end{bmatrix}^{-1} \begin{bmatrix} F^i_T \\ M_{yaw} \\ M_{pitch} \\ M^b_{roll} \end{bmatrix} \tag{3.4}$$

$$\underbrace{\begin{bmatrix} \omega^2_{motor,1} \\ \omega^2_{motor,2} \\ \omega^2_{motor,3} \\ \omega^2_{motor,4} \end{bmatrix}}_{\omega^2_{motor}} = \begin{bmatrix} \omega^2_{motor,front-right} \\ \omega^2_{motor,front-left} \\ \omega^2_{motor,back-right} \\ \omega^2_{motor,back-left} \end{bmatrix} = \begin{bmatrix} m_{11} & m_{12} & m_{13} & m_{14} \\ m_{21} & -m_{22} & m_{23} & -m_{24} \\ m_{31} & -m_{32} & -m_{33} & m_{34} \\ m_{41} & m_{42} & -m_{43} & -m_{44} \end{bmatrix} \begin{bmatrix} F^i_T \\ M^b_{yaw} \\ M^b_{pitch} \\ M^b_{roll} \end{bmatrix} \tag{3.5}$$

The following relation between the motor speeds and forces is found by expanding the above equation. This is in line with what was discussed in this section qualitatively.

$$\begin{aligned} \omega^2_{motor,front-right} &= m_{11} F^i_T + m_{12} M^b_{yaw} + m_{13} M^b_{roll} + m_{14} M^b_{pitch} \\ \omega^2_{motor,front-left} &= m_{21} F^i_T - m_{22} M^b_{yaw} + m_{23} M^b_{roll} - m_{24} M^b_{pitch} \\ \omega^2_{motor,back-right} &= m_{31} F^i_T - m_{32} M^b_{yaw} - m_{33} M^b_{roll} + m_{34} M^b_{pitch} \\ \omega^2_{motor,back-left} &= m_{41} F^i_T + m_{42} M^b_{yaw} - m_{43} M^b_{roll} - m_{44} M^b_{pitch} \end{aligned} \tag{3.6}$$

By considering that the actual controller commands (defined as $\begin{bmatrix} f_{T_{controller}} \\ \tau_{yaw_{controller}} \\ \tau_{pitch_{controller}} \\ \tau_{roll_{controller}} \end{bmatrix}$) are proportional to the thrust forces and moments applied on the drone with a constant gain of $K_{force2cmd}$[2], the controller commands will be:

$$\begin{bmatrix} f_{T_{controller}} \\ \tau_{yaw_{controller}} \\ \tau_{pitch_{controller}} \\ \tau_{roll_{controller}} \end{bmatrix} = K_{force2cmd}\text{sign}(\omega_{motor}) \begin{bmatrix} F_T^i \\ M_{yaw}^b \\ M_{pitch}^b \\ M_{roll}^b \end{bmatrix} \tag{3.7}$$

Where $\text{sign}(\omega_{motor}) = \begin{bmatrix} 1 & -1 & 1 & -1 \end{bmatrix}$ shows the rotation angle directions based on the right hand rule. As a result, above equation becomes:

$$\begin{bmatrix} f_{T_{controller}} \\ \tau_{yaw_{controller}} \\ \tau_{pitch_{controller}} \\ \tau_{roll_{controller}} \end{bmatrix} = K_{force2cmd}\begin{bmatrix} 1 & -1 & 1 & -1 \end{bmatrix} \begin{bmatrix} F_T^i \\ M_{yaw}^b \\ M_{pitch}^b \\ M_{roll}^b \end{bmatrix} \tag{3.8}$$

The substitution of Equation (3.3) into Equation (3.7) gives:

[2]This constant gain maps the controller commands to the accepted commands accepted by the speed controller of the motor. This normalization is done to prevent overly large gains for the controllers.

$$\begin{bmatrix} f_{T_{controller}} \\ \tau_{yaw_{controller}} \\ \tau_{pitch_{controller}} \\ \tau_{roll_{controller}} \end{bmatrix} = K_{force2cmd}\ \rho A_{prop} R_{prop}^2$$

$$\begin{bmatrix} 1 & -1 & 1 & -1 \end{bmatrix} \begin{bmatrix} C_{F_{T_1}} & C_{F_{T_2}} & C_{F_{T_3}} & C_{F_{T_4}} \\ lC_{F_{T_1}} sin(a_1) & lC_{F_{T_2}} sin(a_2) & lC_{F_{T_3}} sin(a_3) & lC_{F_{T_4}} sin(a_4) \\ -lC_{F_{T_1}} cos(a_1) & -lC_{F_{T_2}} cos(a_2) & -lC_{F_{T_3}} cos(a_3) & -lC_{F_{T_4}} cos(a_4) \\ R_{prop}C_{M_{D_1}} & -R_{prop}C_{M_{D_2}} & R_{prop}C_{M_{D_3}} & -R_{prop}C_{M_{D_4}} \end{bmatrix} \underbrace{\begin{bmatrix} \omega_{motor,1}^2 \\ \omega_{motor,2}^2 \\ \omega_{motor,3}^2 \\ \omega_{motor,4}^2 \end{bmatrix}}_{\omega_{motor}^2} \tag{3.9}$$

It is also important to remember that the values calculated from Equation (3.9) must always be between the minimum and maximum controller commands that are possible (which are related to the maximum and minimum possible motor speed). This means if the commands go out of the bounds, they are saturated to the bounds.

Inverting equation Equation (3.9) gives motor speeds in terms of the controller commands[3]:

[3]This equation is always invertible unless the motor speeds go to infinity, which is impossible due to the physical limitations set for motors.

$$\begin{bmatrix} \omega^2_{motor,1} \\ \omega^2_{motor,2} \\ \omega^2_{motor,3} \\ \omega^2_{motor,4} \end{bmatrix} = \frac{1}{K_{force2cmd}\ \rho A_{prop} R^2_{prop}}$$
$$\left(\begin{bmatrix} 1 & -1 & 1 & -1 \end{bmatrix} \begin{bmatrix} C_{F_{T_1}} & C_{F_{T_2}} & C_{F_{T_3}} & C_{F_{T_4}} \\ lC_{F_{T_1}} sin(a_1) & lC_{F_{T_2}} sin(a_2) & lC_{F_{T_3}} sin(a_3) & lC_{F_{T_4}} sin(a_4) \\ -lC_{F_{T_1}} cos(a_1) & -lC_{F_{T_2}} cos(a_2) & -lC_{F_{T_3}} cos(a_3) & -lC_{F_{T_4}} cos(a_4) \\ R_{prop} C_{M_{D_1}} & -R_{prop} C_{M_{D_2}} & R_{prop} C_{M_{D_3}} & -R_{prop} C_{M_{D_4}} \end{bmatrix} \right)^{-1} \begin{bmatrix} f_{T_{controller}} \\ \tau_{yaw_{controller}} \\ \tau_{pitch_{controller}} \\ \tau_{roll_{controller}} \end{bmatrix} \tag{3.10}$$

3.3 Controllers

In this research, a cascade controlling approach is used [17, 51, 52, 53, 54].

The solution to an ordinary differential equation (ODE) of order two converges to zero if the coefficients are positive. By using this concept, if it is considered that the position errors and yaw error follow a second-order ordinary differential equation with positive coefficients, then the errors can converge to zero when the proper gains are selected [52, 55]. This is formulated as:

$$\begin{aligned} \ddot{e}_r(t) + K_{D,r}\dot{e}_r(t) + K_{P,r}e_r(t) = 0 \qquad & K_{D,r}, K_{P,r} > 0 \\ \ddot{e}_{\theta_3}(t) + K_{D,\theta_3}\dot{e}_{\theta_3}(t) + K_{P,\theta_3}e_{\theta_3}(t) = 0 \qquad & K_{D,\theta_3}, K_{P,\theta_3} > 0 \end{aligned} \tag{3.11}$$

Substitution of Equation (3.2) into Equation (3.11) gives Equation (3.12).

$$\begin{gathered}(\ddot{r}_{des}(t) - \ddot{r}(t)) + K_{D,r}\left(\dot{r}_{des}(t) - \dot{r}(t)\right) + K_{P,r}\left(r_{des}(t) - r(t)\right) = 0 \\ \left(\ddot{\theta}_{3,des}(t) - \ddot{\theta}_3(t)\right) + K_{D,\theta_3}\left(\dot{\theta}_{3,des}(t) - \dot{\theta}_3(t)\right) + K_{P,\theta_3}\left(\theta_{3,des}(t) - \theta_3(t)\right) = 0\end{gathered} \tag{3.12}$$

Expanding the first vector equation into its elements gives:

$$\begin{aligned}(\ddot{x}_{des} - \ddot{x}) + K_{D,x}\left(\dot{x}_{des} - \dot{x}\right) + K_{P,x}\left(x_{des} - x\right) &= 0 \\ (\ddot{y}_{des} - \ddot{y}) + K_{D,y}\left(\dot{y}_{des} - \dot{y}\right) + K_{P,y}\left(y_{des} - y\right) &= 0 \\ (\ddot{z}_{des} - \ddot{z}) + K_{D,z}\left(\dot{z}_{des} - \dot{z}\right) + K_{P,z}\left(z_{des} - z\right) &= 0 \\ \left(\ddot{\theta}_{3,des} - \ddot{\theta}_3\right) + K_{D,\theta_3}\left(\dot{\theta}_{3,des} - \dot{\theta}_3\right) + K_{P,\theta_3}\left(\theta_{3,des} - \theta_3\right) &= 0\end{aligned} \tag{3.13}$$

To satisfy Equation (3.12), four different controllers are considered [52, 55]. The closed-loop controller is shown in Figure 3.7. As shown, it consists of two independent controllers that are the height controller and the yaw controller. x and y are controlled by using two chained rows of controllers (cascade controllers). In the following subsections, the details of these controllers are discussed.

3.3.1 Height Controller

The thrust command ($F_{aero,z}$) can be used to control the height of the drone, as shown in Figure 3.8.

Inside the height controller, a PD controller is implemented that follows

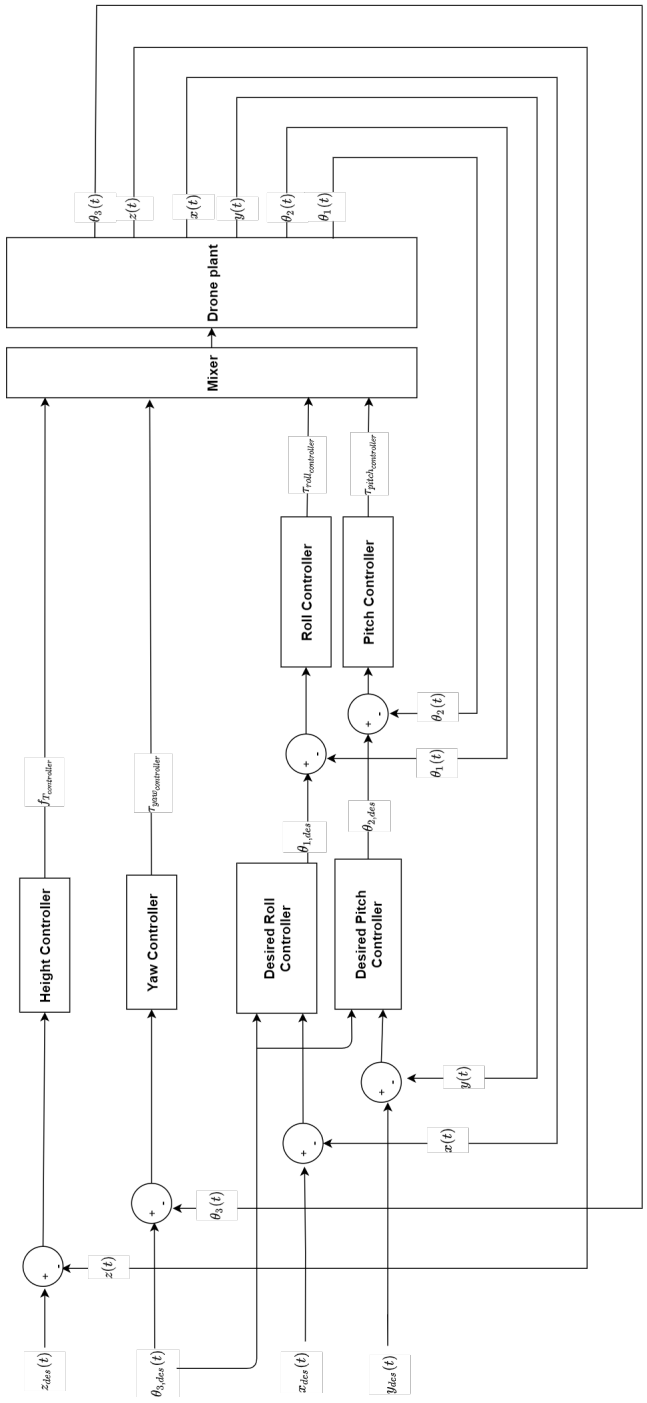

Figure 3.7: The closed-loop controller. In this figure, the drone plant includes the sensors and the onboard estimator.

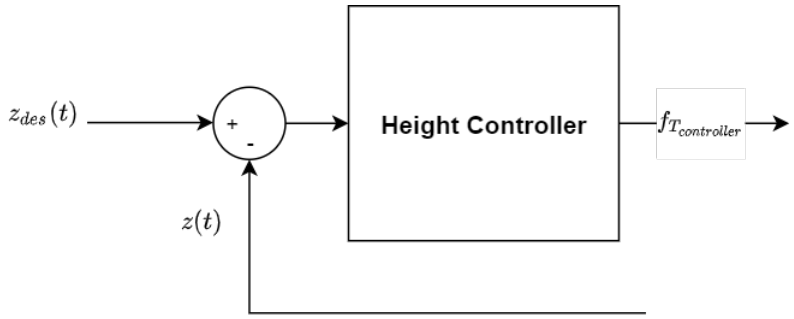

Figure 3.8: Height Controller

Equation (3.14) [52].

$$f_{T_{controller}} = g + \ddot{z}_{des} + K_{d,z}(\dot{z}_{des} - \dot{z}) + K_{p,z}(z_{des} - z) \tag{3.14}$$

where $f_{T_{controller}}$ is the thrust control command.

3.3.2 Yaw Controller

The yaw command (M_{yaw}) can be used to control the yaw of the drone, as shown in Figure 3.9.

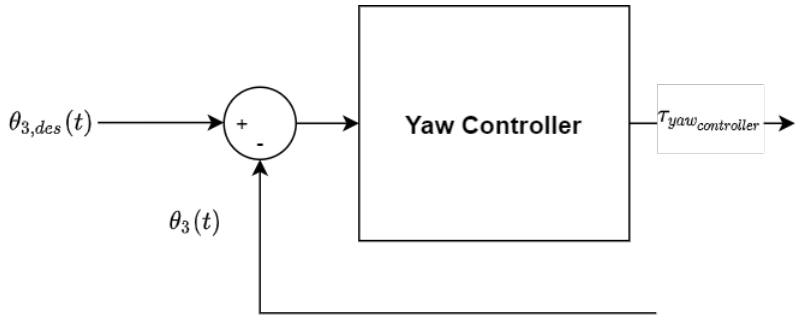

Figure 3.9: Yaw Controller

Inside the yaw controller, a PD controller with the following equation is

implemented [52]:

$$\tau_{yaw_{controller}} = \ddot{\theta}_{3,des} + K_{d,\theta_3}(\dot{\theta}_{3,des} - \dot{\theta}_3) + K_{p,\theta_3}(\theta_{3,des} - \theta_3) \tag{3.15}$$

where $\tau_{yaw_{controller}}$ is the yaw control command.

3.3.3 x and y Control

A combination of roll and pitch commands is used to control its x, y position. To achieve this, a cascade control configuration is used Figure 3.10. The first controller generates the desired roll $\theta_{1,des}$ and pitch $\theta_{2,des}$ that gives us the desired position x_{des}, y_{des}. Then the second controller tries to control roll θ_1 and pitch θ_2 such that these desired roll $\theta_{1,des}$ and pitch $\theta_{2,des}$ are achieved, which ultimately results in the desired positions x_{des}, y_{des}.

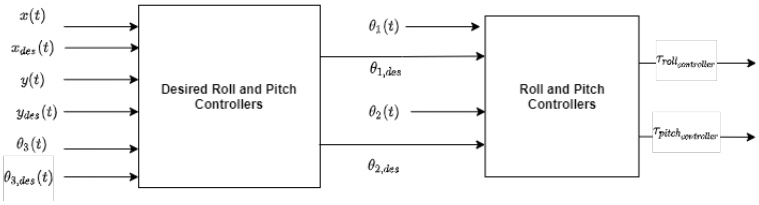

Figure 3.10: Cascade controllers

The control rules for the desired roll and pitch controller are as follows [52]:

$$\theta_{1,des} = \frac{1}{g}([K_{d,x}(\dot{x}_{des} - \dot{x}) + K_{p,x}(x_{des} - x)]\ sin(\theta_{3,des}) - [K_{d,y}(\dot{y}_{des} - \dot{y}) + K_{p,y}(y_{des} - y)]\ cos(\theta_{3,des})) \tag{3.16}$$

$$\theta_{2,des} = \frac{1}{g}([K_{d,x}(\dot{x}_{des} - \dot{x}) + K_{p,x}(x_{des} - x)]\ cos(\theta_{3,des}) + [K_{d,y}(\dot{y}_{des} - \dot{y}) + K_{p,y}(y_{des} - y)]\ sin(\theta_{3,des})) \tag{3.17}$$

In the above equations, the desired yaw angle ($\theta_{3,des}$) is used to map the body fixed x^b, y^b axes to inertial x, y axes. The block diagrams for the desired roll and pitch controllers are shown in Figure 3.11.

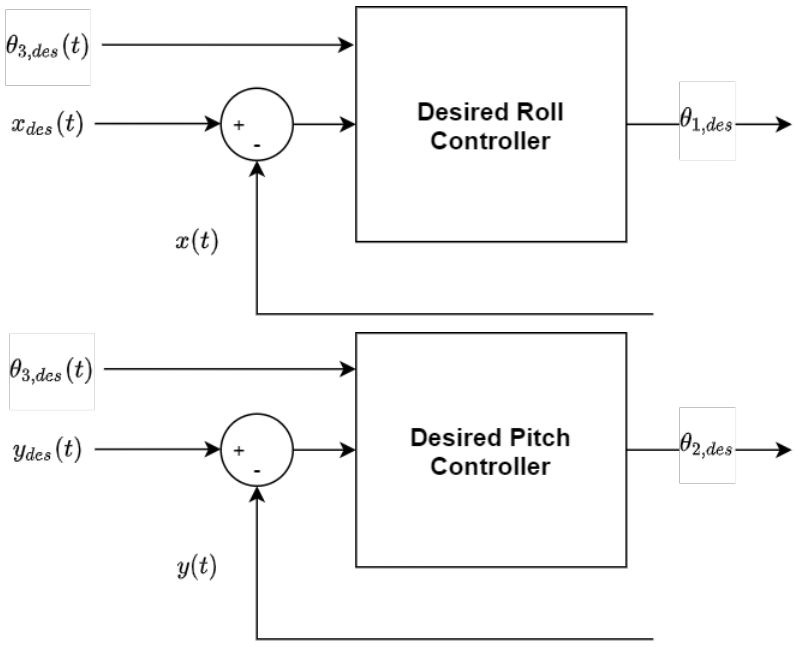

Figure 3.11: Desired roll and pitch controllers

Two PD controllers are implemented to control the roll and pitch based on the calculated desired roll and desired pitch:

$$\tau_{roll_{controller}} = \ddot{\theta}_{1,des} + K_{d,\theta_1}(\dot{\theta}_{1,des} - \dot{\theta}_1) + K_{p,\theta_1}(\theta_{1,des} - \theta_1) \tag{3.18}$$

$$\tau_{pitch_{controller}} = \ddot{\theta}_{2,des} + K_{d,\theta_2}(\dot{\theta}_{2,des} - \theta_2) + K_{p,\theta_2}(\theta_{2,des} - \theta_2) \tag{3.19}$$

where $\tau_{roll_{controller}}$ and $\tau_{pitch_{controller}}$ are the roll and pitch control commands.

The block diagrams for these roll and pitch controllers are shown in Figure 3.12.

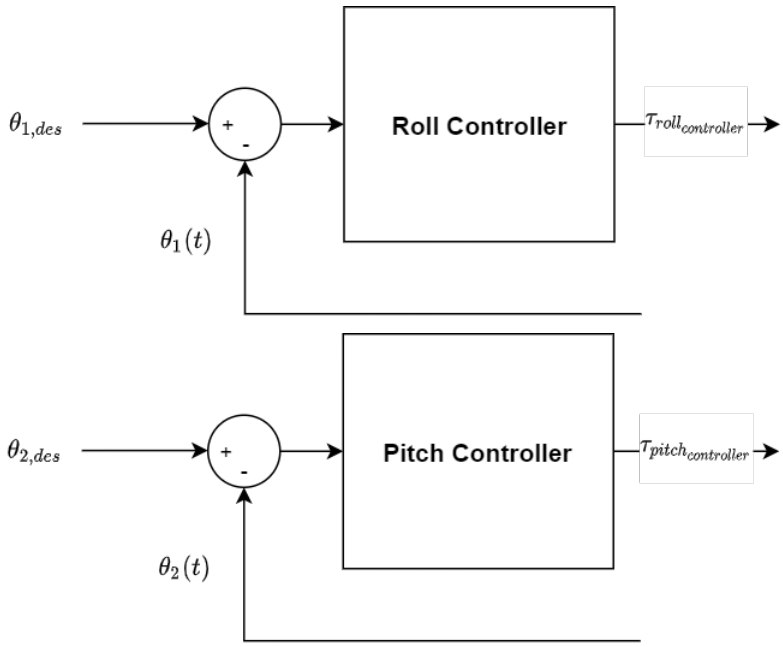

Figure 3.12: Roll and pitch controllers

In conclusion, as shown in Figure 3.7, each of these controllers is put

together to make a complex system of closed-loop controllers that control the drone.

3.3.4 Landing logic

The onboard landing logic is responsible for reducing the altitude at the end of the flight, and it is triggered once the landing time is reached. The logic decreases z_{des} linearly over time, and the closed-loop controllers adjust the motor commands based on this changing z_{des}. Once the drone reaches near the ground, the motors are turned off.

$$z_{des} = z_{des-initial} - K_{landing}(t - t_{landing}) \tag{3.20}$$

where $K_{landing}$ specifies the speed of reducing the altitude, and it was considered to be the same as $T_{sampling}$ in this research.

3.4 Closed-loop control results

Different configurations were tested to verify the closed-loop controller functionality, two of which are presented in the following.[4]

The controller gains are shown in Table 3.1.

[4]The results that are presented in this chapter are related to the Parrot Mambo drone, which its physical properties are given in Section §A. These results can be extended to other types of drones and configurations.

Table 3.1: Controller gains

Gain	value	Gain	value
$K_{p,x}$	-0.24	K_{p,θ_1}	0.014
$K_{d,x}$	0.1	K_{d,θ_1}	0.002
$K_{i,x}$	0	K_{i,θ_1}	0.01
$K_{p,y}$	-0.24	K_{p,θ_2}	0.01
$K_{d,y}$	-0.1	K_{d,θ_2}	0.0028
$K_{i,y}$	0	K_{i,θ_2}	0.01
$K_{p,z}$	1.5	K_{p,θ_3}	0.005
$K_{d,z}$	0.3	K_{d,θ_3}	0.0012
$K_{i,z}$	0.05	K_{i,θ_3}	0

The first flight results are related to an autonomous hover flight with the desired height of 1 m and automatic take-off and landing. Figure 3.13 shows the altitude of the drone during this flight over time. As shown, the drone successfully takes off and hovers around the desired height and finally lands on the ground according to the landing logic discussed in 3.3.4.

The error in the z direction is plotted in Figure 3.14, which is converged to 0 after the initial overshoot.

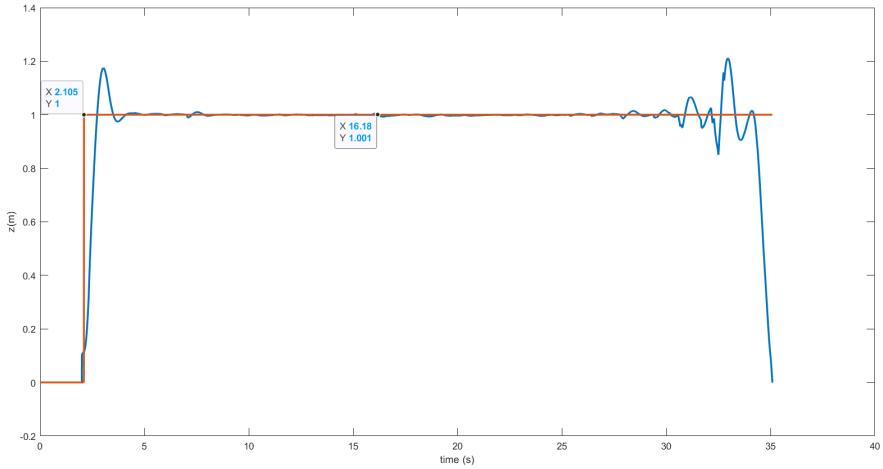

Figure 3.13: Parrot Mambo altitude (z) in a hover flight with the desired height of 1 m

The velocity of the drone in the z direction is shown in Figure 3.15. As seen, the velocity of the drone increases to reach the altitude of 1 m. Then the velocity goes to 0 for the duration of the first hover. In the end, the velocity becomes negative to support decreasing the height.

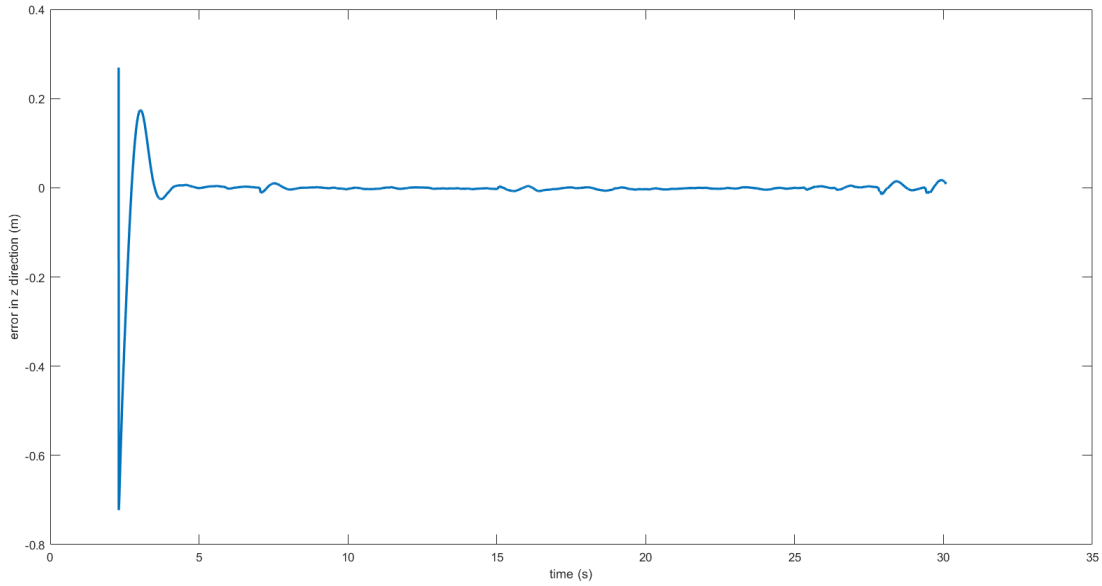

Figure 3.14: Parrot Mambo error in the z direction in a hover flight with the desired height of 1 m

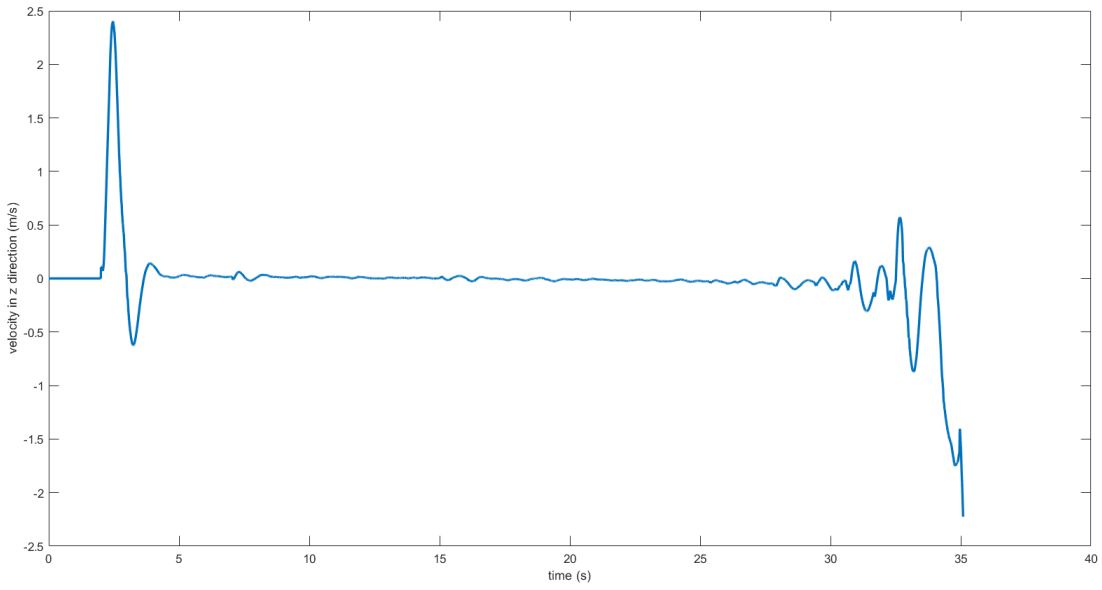

Figure 3.15: Parrot Mambo velocity in the z direction in a hover flight with the desired height of 1 m

The closed-loop command sent to motor 3 of the drone is shown in Figure 3.16. The command for motor 1 is similar to motor 3. For motor 2 and motor 4, the commands are like a mirror of the command for the other two motors, as shown in Figure 3.17.

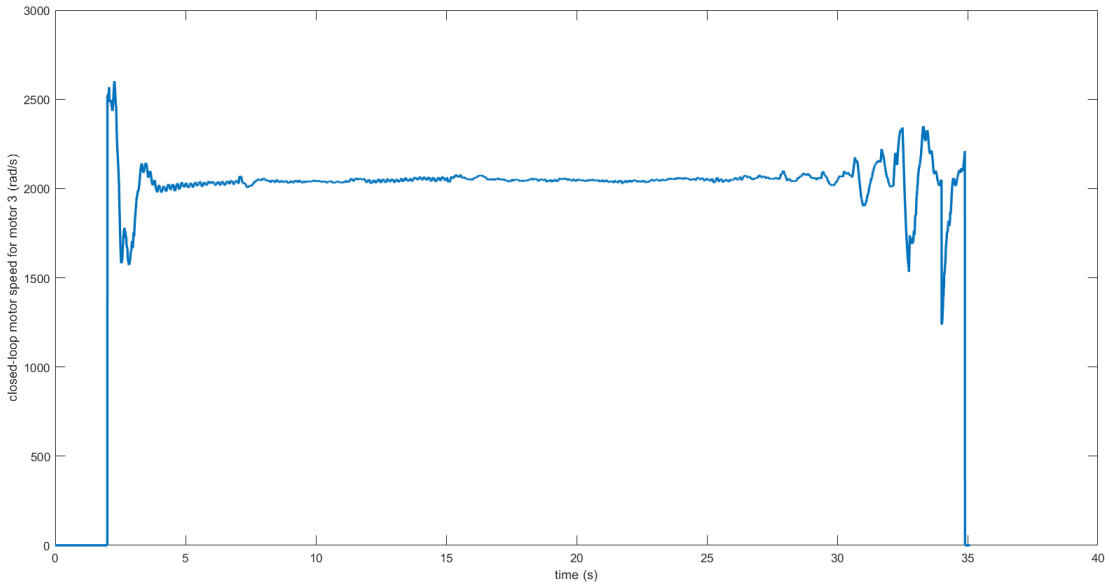

Figure 3.16: The closed-loop command sent to motor 3 in a hover flight with the desired height of 1 m.

In conclusion, this flight shows that the controller successfully tracks the take-off logic, the hovering flight at the altitude of 1 m, and the landing logic.

The next flight shows a hover flight in which the desired altitude changes in the middle of hovering from 1 m to 2 m. The drone's position in the z direction is shown in Figure 3.18, which demonstrates proper position tracking

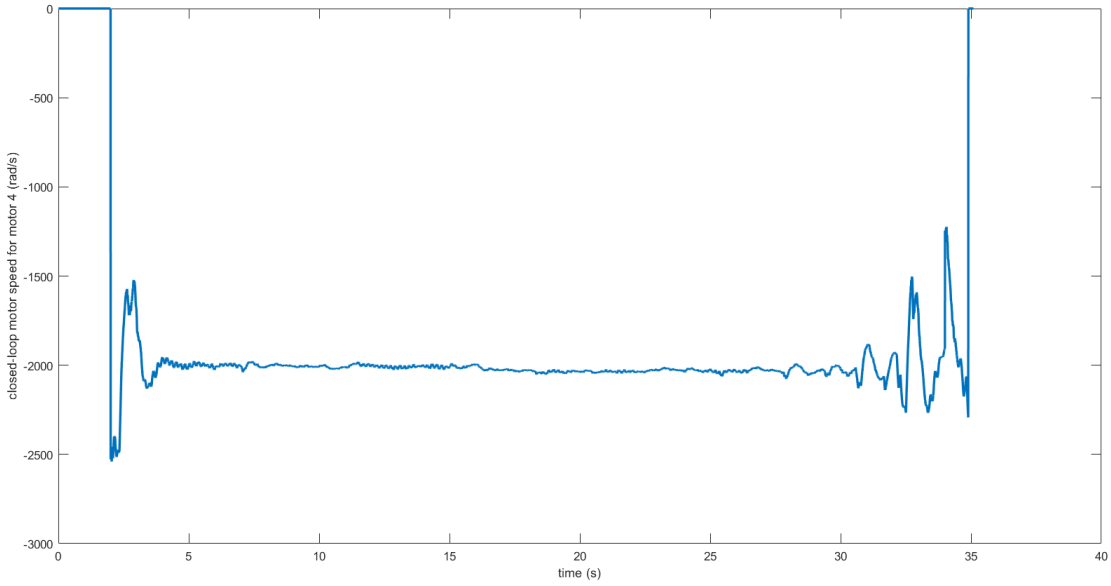

Figure 3.17: The closed-loop command sent to motor 4 in a hover flight with the desired height of 1 m.

with an error of less than 5 cm after the drone reaches the stable altitude. The minor hiccups in the chart are due to disturbances of the real environment, the inaccuracy of the motors and sensors, and the noise of the flight data.

The velocity of the drone in the z direction is plotted in Figure 3.19. As seen, the velocity of the drone increases to reach the altitude of 1 m. Then the velocity goes to 0 for the duration of the first hover. Then the velocity again increases to get the drone to the altitude of 2 m and then decreases to 0 for the period of the second hover. In the end, the velocity becomes negative to support reducing the height and finally goes to 0 for a proper

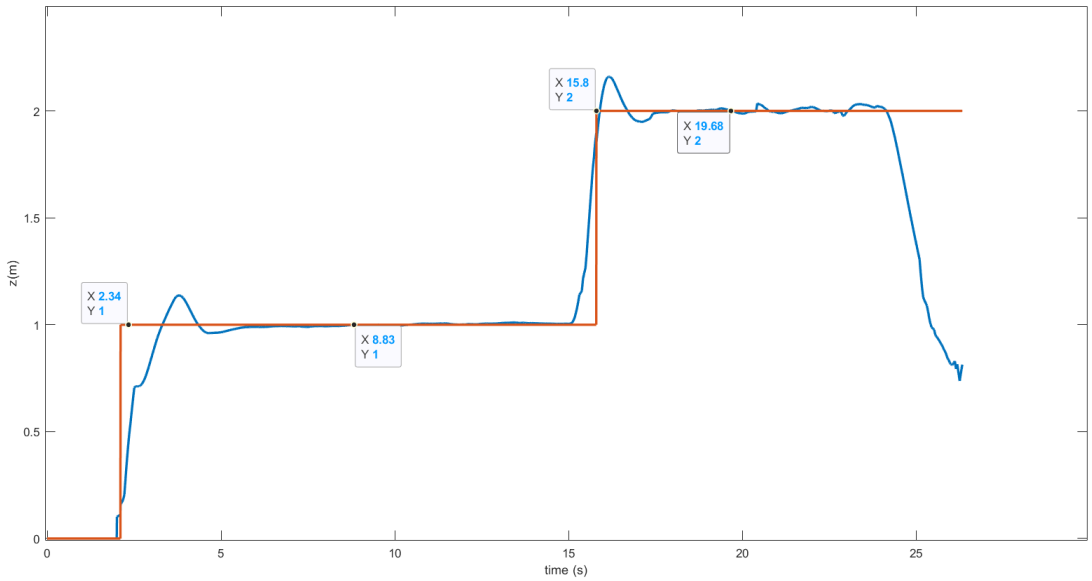

Figure 3.18: Parrot Mambo altitude (z) in a hover flight with the variable desired height of 1 m to 2 m.

landing.

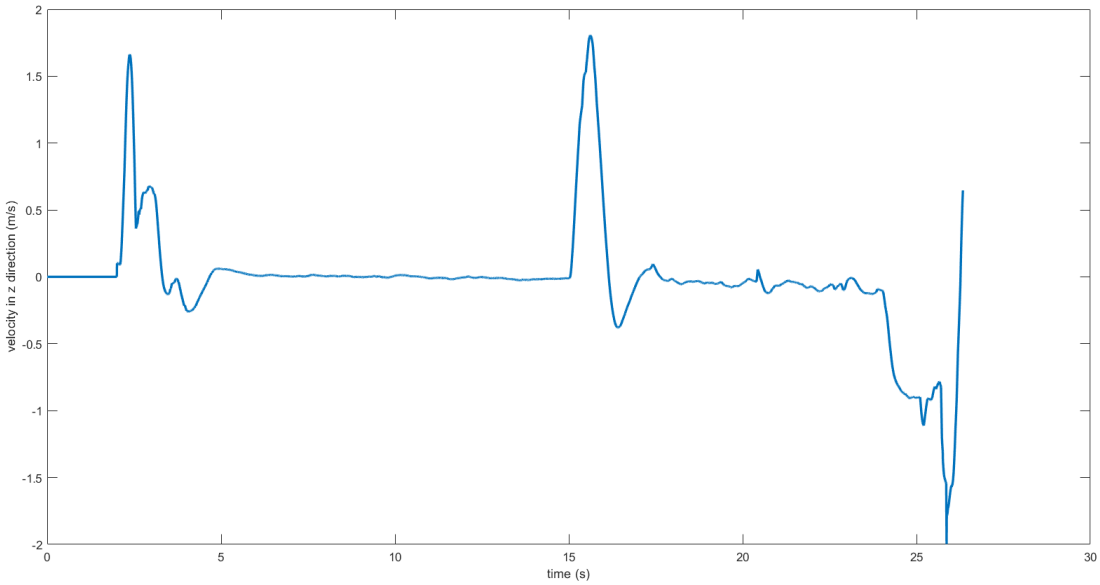

Figure 3.19: Parrot Mambo velocity in the z direction in a hover flight with the variable desired height of 1 m to 2 m.

The closed-loop command sent to motor 3 and motor 4 of the drone are shown in Figure 3.16 and Figure 3.21 respectively. As expected, the command for motor 1 is similar to motor 3. For motor 2 and motor 4, the commands are like a mirror of the command for the other two motors.

In conclusion, this flight shows that the controller successfully tracks the take-off logic, hover flight at the altitude of 1 m, changing the height from 1 m to 2m, the hovering flight at the altitude of 2 m, and the landing logic as it was set in the onboard software.

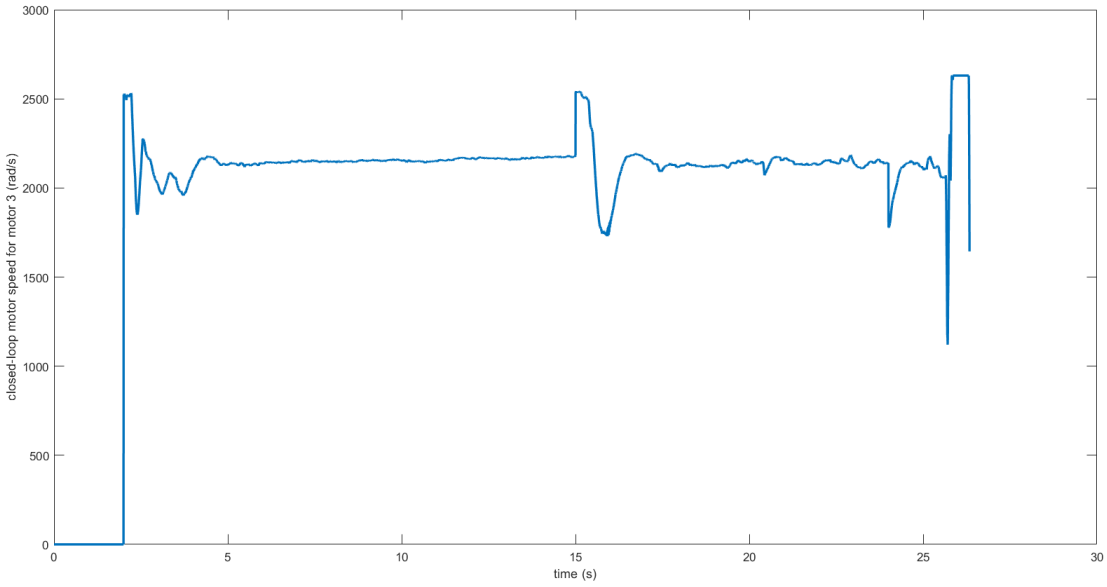

Figure 3.20: The closed-loop command sent to motor 3 in a hover flight with the variable desired height of 1 m to 2 m.

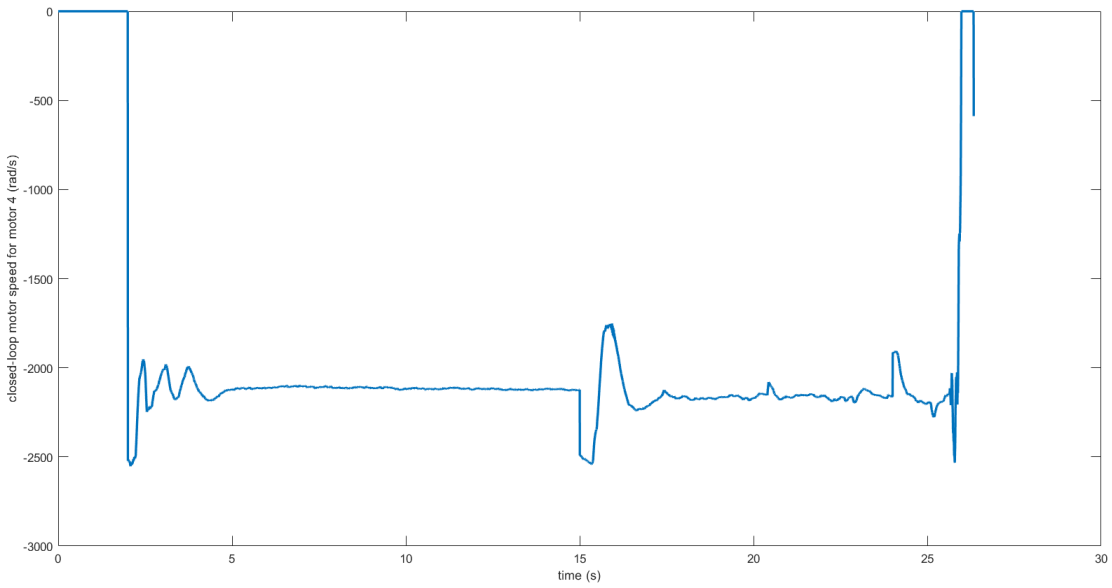

Figure 3.21: The closed-loop command sent to motor 4 in a hover flight with the variable desired height of 1 m to 2 m.

Chapter 4

Feed-forward Drone Control

As discussed in 1.1.1, the simulation of virtual environments requires a feed-forward controller that compensates for the dynamics of the drone. This compensation needs the dynamics of the drone to be reversed, which means that these controllers need to be preplanned using the dynamic model that is known for the drone, and they cannot use the real-time data of the sensors.

Figure 4.1 shows the feed-forward controller and the closed-loop controller in relation to the drone plant. As seen, the feed-forward controller output is directly fed to the drone plant, and this output is calculated without getting any feedback from the sensors.

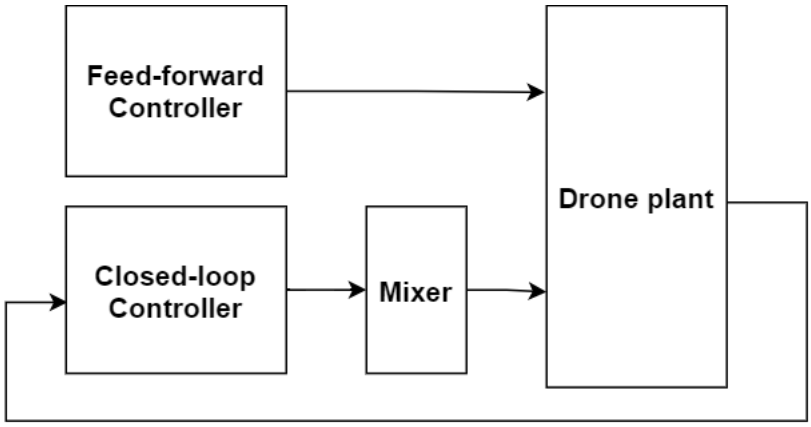

Figure 4.1: The feed-forward controller and the closed-loop controller. In this figure, the drone plant includes the sensors and the onboard estimator.

To understand this, consider a simple case of the motion for z translation of a simple point mass under the influence of a vertical force. Using Newton's equation gives:

$$F_{feed-forward} - mg = ma \tag{4.1}$$

To emulate different gravity, a goal for the right-hand side of the equation should be set. For example, to simulate a zero-gravity environment, it is considered that the mass floats, and so its acceleration in the vertical direction should be 0. This will make Equation (4.1) like the following:

$$F_{feed-forward} - mg = 0 \tag{4.2}$$

So the preplanned feed-forward forward force will be:

$$F_{feed-forward} = mg \tag{4.3}$$

As it is apparent in Equation (4.3), the proper feed-forward controller is directly dependent on mg, which defines the native dynamics of the system. So it is clear that to do precise feed-forward dynamic emulation, knowledge of the system dynamics is necessary. The dynamics here depend on the accurate understanding of mg. Reversing these native dynamics gives the desired feed-forward force.

In practice, the feed-forward forces are a function of the drone's parameters, such as motor constants. This requires calculating the control signals in reverse based on the desired feed-forward force and the parameters. The control signals are the motor speeds, and since the motor speeds generate the thrust forces, to achieve a desired feed-forward control goal, the motor speeds should be calculated based on the plan. Here, it is considered that the goal is to reduce the external environment's effect (such as gravity). In this research, the equations are discussed in a general form, and later in Section §7.1, two specific cases (50% reduced gravity, and 80% reduced gravity) are discussed.

4.1 Aerodynamic force (F_{aero})

To remove the effect of the external environment, the total external force that is applied on the drone (F_{tot}) needs to be zero. So, from Equation (2.29), the following can be written:

$$F_{tot} = 0 \rightarrow RF_{\text{aero}} - mg\hat{k} = 0 \tag{4.4}$$

This means that the desired aerodynamic force (generated by the feed-

forward controller) should be[1]:

$$F_{\text{aero}} = R^{-1} mg\hat{k} \tag{4.5}$$

The above equation can be expanded to its elements as:

$$F_{aero} = \begin{bmatrix} F_{aero,x} \\ F_{aero,y} \\ F_{aero,z} \end{bmatrix} = mgR^{-1} \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \tag{4.6}$$

which is then simplified to:

$$F_{aero} = \begin{bmatrix} F_{aero,x} \\ F_{aero,y} \\ F_{aero,z} \end{bmatrix} = mg \begin{bmatrix} s_1 s_3 + c_1 c_3 s_2 \\ c_1 s_2 s_3 - c_3 s_1 \\ c_1 c_2 \end{bmatrix} \tag{4.7}$$

To design a feed-forward controller, the controller commands should be extracted from Equation (4.7). This means that the left-hand side of the equation should be modelled as a function of the controller signals (motor speeds) and then solved in terms of these control signals.

In a case where the drone flies near the hover configuration, θ_1 and θ_2 will be small, which means $s_1 \approx 0$ and $s_2 \approx 0$. In this case, the above equation can be simplified to:

$$\begin{bmatrix} F_{aero,x} \\ F_{aero,y} \\ F_{aero,z} \end{bmatrix} = mg \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \tag{4.8}$$

[1]Since R is orthonormal, this inversion step is always possible, and the inverse is equal to its transpose.

That means near hover configuration, the required aerodynamic force for the feed-forward controller is equal to the drone's weight.

$$F_{aero,z} = mg \tag{4.9}$$

To design a feed-forward controller for the flights that are not near to hover state, the aerodynamic forces in the drone's plane ($F_{aero,x}$ and $F_{aero,y}$) should be modelled and solved in terms of the controller signals.

4.2 Thrust force (F_T)

To find aerodynamic forces that meet the needs of the feed-forward controller, it is needed to calculate the respective thrust forces. From Equation (2.35), the following can be written:

$$F_{aero} = \begin{bmatrix} F_{aero,x} \\ F_{aero,y} \\ F_{aero,z} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ \sum_{n=1}^{N_{motor}} F_{T_n} \end{bmatrix} \tag{4.10}$$

or by only considering the forces in the z direction:

$$F_{aero,z} = \sum_{n=1}^{N_{motor}} F_{T_n} \tag{4.11}$$

By assuming that the thrust forces generated by each motor are equal to each other, each thrust force will be

$$F_{T_n} = \frac{F_{aero,z}}{N} \tag{4.12}$$

Near the hovering flight, according to Equation (4.9), the thrust forces for each motor that reduce the effect of gravity should be:

$$F_{T_n} = \frac{mg}{N} \tag{4.13}$$

According to the aerodynamic equation of the thrust force (Equation (2.31)), the thrust force could be written for all the motors in the following matrix format:

$$F_T = \rho A_{prop} R_{prop}^2 C_{F_T} \omega_{motor}^2 \tag{4.14}$$

Separating motor speeds from this equation gives:

$$\omega_{motor} = \text{sign}(\omega_{motor}) \sqrt{(\rho A_{prop} R_{prop}^2 C_{F_T})^{-1} F_T} \tag{4.15}$$

Where $\text{sign}(\omega_{motor}) = \begin{bmatrix} 1 & -1 & 1 & -1 \end{bmatrix}$. So, the vector form of this equation becomes:

$$\omega_{motor} = \begin{bmatrix} \sqrt{(\rho A_{prop_1} R_1^2 C_{F_{T_1}})^{-1} F_{T_1}} \\ -\sqrt{(\rho A_{prop_2} R_2^2 C_{F_{T_2}})^{-1} F_{T_2}} \\ \sqrt{(\rho A_{prop_3} R_3^2 C_{F_{T_2}})^{-1} F_{T_3}} \\ -\sqrt{(\rho A_{prop_4} R_4^2 C_{F_{T_3}})^{-1} F_{T_4}} \end{bmatrix} \tag{4.16}$$

As you see, in the general case, the thrust coefficients and radius for each different motor can be different. By considering that $\rho A_{prop_1} R_{prop}^2$ is equal for all the motors, Equation (4.16) is simplified to:

$$\omega_{motor} = \frac{1}{\sqrt{\rho A_{prop_1} R_{prop}^2}} \begin{bmatrix} \sqrt{\frac{F_{T_1}}{C_{F_{T_1}}}} \\ -\sqrt{\frac{F_{T_2}}{C_{F_{T_2}}}} \\ \sqrt{\frac{F_{T_3}}{C_{F_{T_3}}}} \\ -\sqrt{\frac{F_{T_4}}{C_{F_{T_4}}}} \end{bmatrix} \tag{4.17}$$

In case it is considered that the coefficients of the thrust are the same for all propellers (which holds if the different motors and propellers are the same model), C_{F_T} can be factored out from the above equation, and the following is achieved, in which the two vectors are multiplied element-wise using a dot product.

$$\omega_{motor} = \frac{1}{\sqrt{\rho A_{prop_1} R_{prop}^2}} \frac{1}{\sqrt{C_{F_T}}} \left(\begin{bmatrix} 1 \\ -1 \\ 1 \\ -1 \end{bmatrix} \cdot \sqrt{\begin{bmatrix} F_{T_1} \\ F_{T_2} \\ F_{T_3} \\ F_{T_4} \end{bmatrix}} \right) \tag{4.18}$$

By adding the condition from Equation (4.12), the above equation is simplified to

$$\omega_{motor} = \frac{\sqrt{\frac{F_{aero,z}}{N}}}{\sqrt{\rho A_{prop_1} R_{prop}^2 C_{F_T}}} \begin{bmatrix} 1 \\ -1 \\ 1 \\ -1 \end{bmatrix} \tag{4.19}$$

This gives the motor speed required for each rotor as:

$$\omega_{motor_i} = \text{sign}(\omega_{motor_i}) \frac{\sqrt{\frac{F_{aero,z}}{N}}}{\sqrt{\rho A_{prop_1} R_{prop}^2 C_{F_T}}} \qquad i = 1, 2, 3, 4 \tag{4.20}$$

Based on Equation (4.9), flying near the hover configuration simplifies Equation (4.19):

$$\omega_{motor} = \frac{\sqrt{\frac{mg}{N}}}{\sqrt{\rho A_{prop_1} R_{prop}^2 C_{F_T}}} \begin{bmatrix} 1 \\ -1 \\ 1 \\ -1 \end{bmatrix} \tag{4.21}$$

4.3 Feed-forward controller commands

If it is needed to compare the feed-forward commands to the closed-loop commands directly, the controller commands related to these motor speeds should be calculated. Equation (4.19) can be substituted to Equation (3.9):

$$\begin{bmatrix} f_{T_{controller}} \\ \tau_{yaw_{controller}} \\ \tau_{pitch_{controller}} \\ \tau_{roll_{controller}} \end{bmatrix} = K_{force2cmd}\ \rho A_{prop} R_{prop}^2$$

$$\begin{bmatrix} 1 & -1 & 1 & -1 \end{bmatrix} \begin{bmatrix} C_{F_{T_1}} & C_{F_{T_2}} & C_{F_{T_3}} & C_{F_{T_4}} \\ lC_{F_{T_1}} sin(a_1) & lC_{F_{T_2}} sin(a_2) & lC_{F_{T_3}} sin(a_3) & lC_{F_{T_4}} sin(a_4) \\ -lC_{F_{T_1}} cos(a_1) & -lC_{F_{T_2}} cos(a_2) & -lC_{F_{T_3}} cos(a_3) & -lC_{F_{T_4}} cos(a_4) \\ R_{prop} C_{M_{D_1}} & -R_{prop} C_{M_{D_1}} & R_{prop} C_{M_{D_1}} & -R_{prop} C_{M_{D_1}} \end{bmatrix} \frac{\sqrt{\frac{F_{aero,z}}{N}}}{\sqrt{\rho A_{prop} R_{prop}^2 C_{F_T}}} \begin{bmatrix} 1 \\ -1 \\ 1 \\ -1 \end{bmatrix} \tag{4.22}$$

By substituting Equation (4.12) and by assuming equal drag moment coefficients for all the motor, the above equation is further simplified to

$$\begin{bmatrix} f_{T_{controller}} \\ \tau_{yaw_{controller}} \\ \tau_{pitch_{controller}} \\ \tau_{roll_{controller}} \end{bmatrix} = K_{force2cmd}\ \sqrt{\rho A_{prop_1}} R_{prop} \sqrt{C_{F_T}} \sqrt{\frac{F_{aero,z}}{N}}$$
$$\begin{bmatrix} 1 & -1 & 1 & -1 \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 & 1 \\ lsin(a_1) & lsin(a_2) & lsin(a_3) & lsin(a_4) \\ -lcos(a_1) & -lcos(a_2) & -lcos(a_3) & -lcos(a_4) \\ R_{prop}\frac{C_{M_D}}{\sqrt{C_{F_T}}} & -R_{prop}\frac{C_{M_D}}{\sqrt{C_{F_T}}} & R_{prop}\frac{C_{M_D}}{\sqrt{C_{F_T}}} & -R_{prop}\frac{C_{M_D}}{\sqrt{C_{F_T}}} \end{bmatrix} \begin{bmatrix} 1 \\ -1 \\ 1 \\ -1 \end{bmatrix} \tag{4.23}$$

By calculating the matrix multiplications and some simplifications, the above equation is simplified to

$$\begin{bmatrix} f_{T_{controller}} \\ \tau_{yaw_{controller}} \\ \tau_{pitch_{controller}} \\ \tau_{roll_{controller}} \end{bmatrix} = K_{force2cmd}\ \sqrt{\rho A_{prop_1}} R_{prop} \sqrt{C_{F_T}} \sqrt{\frac{F_{aero,z}}{N}} \begin{bmatrix} 0 \\ -lsin(a_1) + lsin(a_2) - lsin(a_3) + lsin(a_4) \\ -lcos(a_1) + lcos(a_2) - lcos(a_3) + lcos(a_4) \\ -4R_{prop}\frac{C_{M_D}}{\sqrt{C_{F_T}}} \end{bmatrix} \tag{4.24}$$

If the drone flies near hover, Equation (4.9) ($F_{aero,z} = mg$) can be substituted into the above equation as

$$\begin{bmatrix} f_{T_{controller}} \\ \tau_{yaw_{controller}} \\ \tau_{pitch_{controller}} \\ \tau_{roll_{controller}} \end{bmatrix} = K_{force2cmd}\ \sqrt{\rho A_{prop_1}} R_{prop} \sqrt{C_{F_T}} \sqrt{\frac{mg}{N}} \begin{bmatrix} 0 \\ -lsin(a_1) + lsin(a_2) - lsin(a_3) + lsin(a_4) \\ -lcos(a_1) + lcos(a_2) - lcos(a_3) + lcos(a_4) \\ -4R_{prop} \frac{C_{M_D}}{\sqrt{C_{F_T}}} \end{bmatrix} \tag{4.25}$$

As discussed throughout this chapter, the motor speeds commanded by the feed-forward controller, which is not a function of the sensor outputs, are calculated from Equation (4.19). This equation is implemented onboard the drone and generates the feed-forward commands fed to the drone plant as shown in Figure 4.1. These commands compensate for the dynamics of the drone, which reduces the work that needs to be done by the closed-loop controller, thus reducing the motor speeds commanded by the closed-loop controller. Later in Section §7.1, the results are discussed.

Chapter 5

System Identification

This chapter aims to determine the drone dynamics with minimal measuring using intelligent system identification techniques. As discussed in 1.1.2, the simulation of virtual environments using drones requires an implementation of a feed-forward controller that compensates for the dynamics of the drone. Feed-forward controllers should be programmed in advance based on a drone model. To do so, the dynamics of the drone should be reversed and then compensated for. This is different from using a closed-loop controller that uses the sensor information in real-time to send proper motor commands to the motors for achieving the control goal. As a result, a high-fidelity model of drone dynamics and aerodynamics is required.

As discussed in Section §1.3 and shown in Equation (4.24), the dynamics of the drone are very complex and depend on many factors such as the motors' thrust force coefficients, the motors' drag moment coefficients, etc. The system of 12 non-linear differential equations prevents us from deriving an analytical solution. Due to all these factors, no direct answer can be

analytically derived, which prevents us from using the usual system identification methods that work based on the analytical answers. As a result, an intelligent method based on data is used to model the drone in this research.

The intelligent system identification method is depicted in Figure 5.1. A gray box model of is system is developed, and in parallel, the actual flight data are collected. Then, the system identifier uses these data to find the parameters of the gray box model that make the errors between the real states and the simulated states go to zero.

Specific data collection tests are designed to measure the inputs and outputs of the actual drone system. By collecting the data from the onboard state estimator, all the states that describe drone dynamics are known. These include position, Euler angles, translational velocities, and angular velocities, which are considered the system's outputs. The model inputs are the commands that are sent to the motors. Then, a drone model is programmed that gets matched to the actual drone by utilizing the collected inputs and outputs in the system identification methods. As shown in Figure 5.1, the system identifier estimates the drone model parameters to minimize the error between the model outputs and drone outputs.

The model obtained from the system identifier is the model that is achieved from the metaheuristic algorithm, which estimates the parameters of the developed gray box drone model.

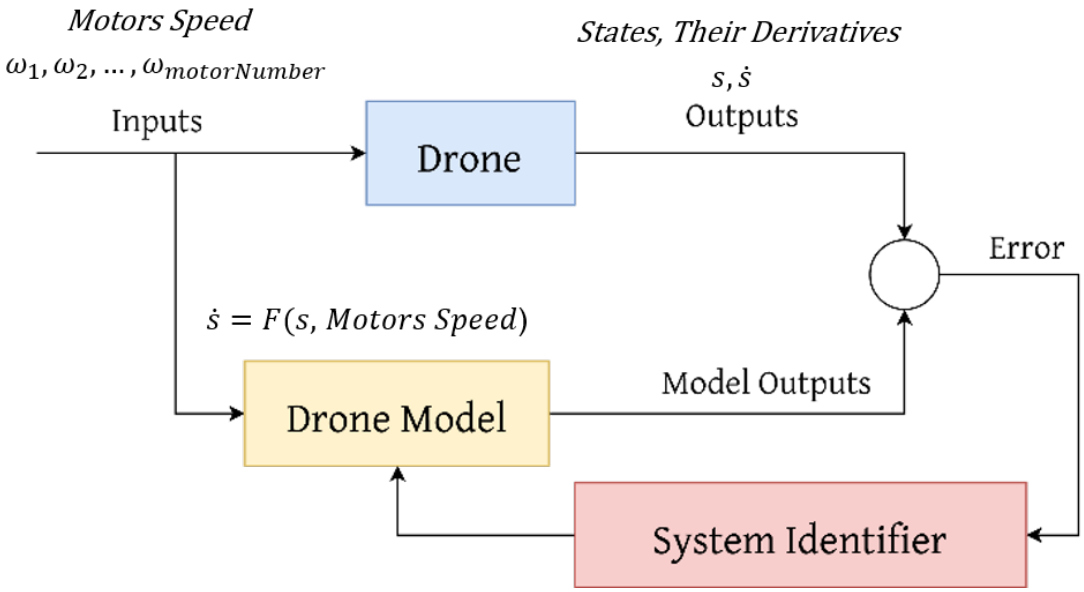

Figure 5.1: System Identification

5.1 Graybox model - Metaheuristic algorithm

This section discusses the gray box model and how a metaheuristic algorithm can be used. In the next section, particle swarm optimization is discussed and implemented as an example of a metaheuristic algorithm.

Rewriting the system identification problem as an optimization method makes it possible to use a metaheuristic algorithm as the system identifier. This algorithm estimates the system's parameters, such as thrust coefficients of the motors, by finding the optimal parameters that make the error between the real experiment's measurements and the model's output minimized.

The metaheuristic algorithms make few or no assumptions about the problem being optimized, and it is able to search a vast answer space. These algorithms do not use the gradient of the problem being optimized, which

means they do not require that the optimization problem be differentiable as is required by classic optimization methods such as gradient descent [56] and quasi-newton [57] methods.

5.1.1 Model parameters

As seen in Equation (4.25), the feed-forward controller needs the following parameters to be known:

- thrust force coefficients of the motors ($C_{F_{T_n}}$)
- drag moment coefficients of the motors ($C_{M_{D_n}}$)
- Propeller radios (R_{prop})
- Area of the propellers (A_{prop})
- Arm length (l)
- drone mass (m)
- air density (ρ)

Among these, the propeller radius (R), area of the propellers (A_{prop}), arm length (l), and drone mass (m) can be measured easily, and the air density (ρ) of the environment is known.[1]

In this research, the model parameters of the algorithm are considered to be the coefficients of thrust ($C_{F_{T_i}}$). In a general case, the coefficient of thrust for each motor is different. The drag moment coefficient is calculated from the following relation that holds for Parrot Mambo:

[1]The values of these parameters for Parrot Mambo are given in Section §A.

$$C_{M_{D_i}} = C_{F_{T_i}} \sqrt{\frac{C_{F_{T_i}}}{2}} = \frac{\sqrt{2C_{F_{T_i}}^3}}{2},\ i = 1, .., n \tag{5.1}$$

For a quadrotor with 4 motors, the number of model parameters is equal to 4. As a result, the thrust force coefficients of the motors ($C_{F_{T_i}}$) need to be estimated by the metaheuristic algorithm. To estimate these parameters from the real flight data, the gray box model of the system (Equation (2.27)) is used by the system identification algorithm.

5.2 Particle Swarm Optimization

Particle swarm optimization (PSO) is the metaheuristic algorithm used in this research to find the optimal parameters based on the defined cost function. The algorithm consists of multiple steps that are repeated until the stopping criteria (e.g. reaching a desired cost function value) are triggered. PSO considers the set of candidate answers in the answer space like a population of the particles that move in the answer space. These particles are randomly distributed in the answer space and then move around to search for the location that minimizes the cost function. The movement of each particle is based on a mathematical formula that depends on the particle's current location (answer), its velocity (variation in the answer), the best location it has seen, and the best location that the whole population has seen so far. This formula has some degree of randomness that prevents it from converging to a local minimum. Because of the communication between the whole population, the movement of each particle is not only around the best answer it has seen itself, but it is also towards the best answer that the whole

population has seen. This swarm-based movement will speed up the process of searching and will result in a better solution compared to an exhaustive search [58].

In the following, each step of the PSO algorithm is discussed in more detail. In this discussion, the drone model parameters (candidate C_{F_T} vectors) are referred to as the locations or parameters of the candidate answers. The answer space is all the possible values of the model parameters, which are considered between 0.001 and 10. The velocity of the candidate answer is the rate at which model parameters change in each step of the algorithm.

5.2.1 Population initialization

The first step of the algorithm is to initialize a population of answers (candidate C_{F_T}) in the answer space. In this research, the candidate answers are generated such that the population follows a random uniform distribution between 0.001 and 10.

5.2.2 Cost function calculation

In this step, the cost function is calculated for each of the particles. The cost function[2] is a function that indicates how close the mathematical model is to the actual system. The cost function can be calculated from the difference between the real drone outputs (target) and the computer model outputs.

The process of calculating this error for each candidate answer consists of multiple steps:

[2]Also called error function, performance index, objective function, competence function, and fit goodness.

1. Getting the current model parameters of the candidate answer: the cost function parameters in the drone model are the C_{F_T}. This step gets the current parameters of the candidate answer. If the algorithm is running for the first time, the parameters are generated randomly in the step discussed in 5.2.1. If the algorithm has been already running, the parameters are calculated based on the rule of PSO (discussed in 5.2.6).

2. Simulation: the computer model outputs must be calculated to find the difference between the actual system outputs (target) and the computer model outputs. Based on Equation (2.27), this model is solved numerically with the collected motor speeds as inputs and the current cost function parameters (model parameters).

$$
\begin{aligned}
\dot{r}^i = \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{z} \end{bmatrix} &= \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{z} \end{bmatrix} \\
\dot{\theta} = \begin{bmatrix} \dot{\theta}_3 \\ \dot{\theta}_2 \\ \dot{\theta}_1 \end{bmatrix} &= \begin{pmatrix} -\frac{c_3}{c_2} & -\frac{s_3}{c_2} & 0 \\ s_3 & -c_3 & 0 \\ -c_3 \tan\theta_2 & -s_3 \tan\theta_2 & -1 \end{pmatrix} \begin{bmatrix} \omega_1 \\ \omega_2 \\ \omega_3 \end{bmatrix} \\
\ddot{r}^i = \begin{bmatrix} \ddot{x} \\ \ddot{y} \\ \ddot{z} \end{bmatrix} &= \frac{1}{m} F^i_{\text{tot}} \\
\dot{\omega}^b = \begin{bmatrix} \dot{\omega}_1 \\ \dot{\omega}_2 \\ \dot{\omega}_3 \end{bmatrix} &= I^{-1} \left(M^b_{\text{tot}} - \omega^b \times I\omega^b \right)
\end{aligned} \tag{5.2}
$$

3. Detecting the outliers: after simulation, if the output of the model has an output that is out of the expected range (the altitude should be between 0 m and 10 m), or the output is not mathematically accepted (the output is infinity or it is not a number), the candidate answer will be assigned an enormous error, and the algorithm will be stopped here for this candidate answer.

4. Data scaling (normalization): because each of the outputs will have a different scale or unit (e.g. position vs angles), the outputs of real system and model should be normalized such that they affect the resulting error equally. This normalization is done for each of the output vectors throughout the time. In this research, max-mean scaling is used, which scales the range of an output vector to the range in $[0, 1]$ based on the maximum and minimum of an output vector.

$$s_{scaled} = \frac{s - min(s)}{max(s) - min(s)} \tag{5.3}$$

5. Calculating the output error: the error between the scaled outputs of the model ($s_{model_{scaled}}$) and the real system ($s_{real_{scaled}}$) is calculated.

$$E_{output} = s_{model_{scaled}} - s_{real_{scaled}} \tag{5.4}$$

Since this error is a vector, a single number needs to be assigned to it, so it is possible to compare different candidate answers. In this research,

the mean of the absolute of the error is used, which is calculated from

$$J_{output} = average(abs(E_{output})) \tag{5.5}$$

Which can be expanded for its elements as

$$J_{output} = \frac{1}{12}\sum_{i=1}^{12} |s_{i,model_{scaled}} - s_{i,real_{scaled}}| \tag{5.6}$$

6. Calculating the finished time error: the error between the time that the model took to finish ($t_{f,model}$) and the time that the real experiment took ($t_{f,real}$) is calculated in this step. If the simulation model crashes to the ground, the crash time is considered the final simulation time.

$$J_{time} = |t_{f,model} - t_{f,real}| \tag{5.7}$$

7. Calculating the cost value: Finally, the cost value is calculated as the weighted sum of the J_{output} and J_{time} with W_{output} and W_{time} being constant weights that show the effect of each value in the final performance index. These weights normalize the range of each of the cost values such that the final cost value represents all the effects similarly or based on their importance.

$$J = W_{output}J_{output} + W_{time}J_{time} \tag{5.8}$$

The weights in the above example are chosen by qualitatively comparing J_{output} and J_{time} and deciding on the importance of each effect. In

this research, W_{output} is selected as 1, and W_{time} is selected as 3.

5.2.3 Outliers removal

After calculating the cost function for each particle, the particles whose cost value is out of the accepted range are removed. This step has a close relationship with Step 3 off 5.2.2 because the outliers are specified in that step. Since the drone model is sensitive to the model parameters, this step is necessary for its convergence.

5.2.4 Finding the best answers

Then when the outliers are removed, the best location that each particle has seen so far ($x_{local\,best_i}$) is updated. Then, among these bests, the best location of the whole population is chosen as the global best ($x_{global\,best}$).

5.2.5 Updating the velocities

In this step, the velocity (the variation in the location) of each particle is updated based on a defined rule. The rule that is used in this research is as follows:

$$v_i' = W_{inertia} v_i + C_1(x_{local\,best_i} - x_i) + C_2(x_{global\,best} - x_i) \tag{5.9}$$

where x_i is the location of the particle, v_i' is the new velocity of the particle i, v_i is the previous velocity of particle i, $W_{inertia}$ is the current inertia weight, and C_1 and C_2 are uniformly distributed random numbers in the interval $[0, 1]$.

The inertia weight is calculated as a fraction of the $W_{inertia}$ from the previous step. In the first step, $W_{inertia}$ is considered 1.

$$W'_{inertia} = 0.98 * W_{inertia} \tag{5.10}$$

5.2.6 Updating the locations

After calculating the velocity of each particle, the movement of the population occurs, and so the new positions should be calculated from the following formula:

$$x'_i = x_i + \Delta x'_i \tag{5.11}$$

where $\Delta x'_i$ is the velocity of the candidate and shown by v'_i. So, the above can be written as:

$$x'_i = x_i + v'_i \tag{5.12}$$

5.2.7 Stopping criteria

The algorithm is repeated until stopping criteria are triggered. In this research, the number of iterations is used as the stopping criteria. The number of iterations is the number of iterations after which the algorithm should be stopped. In this research, 40 is the number of iterations. After 40 iterations, the global best answer is chosen as the answer to the problem.

5.3 Algorithm performance

The estimated model parameters are compared to the parameters given in the datasheets to verify that they are accurate values. Because the valid values of the parameters are unknown, the values from the PSO algorithm are considered to be more accurate since they are derived from the real-flight data. For further proof, one can measure the accurate parameters manually and compare them with the estimated parameters from PSO.

In this thesis, the algorithm ran on a dataset that was collected from a hover flight.

For this case, the population size (swarm size) is considered to be 100. The stopping criteria of this run were the maximum number of iterations, and it was set to 40 iterations.

It is considered that the coefficients of the thrusts (the model parameters) are in this range.

$$0.001 < C_{F_{T_i}} < 0.1 \tag{5.13}$$

A run of the PSO algorithm shows that it can predict the values of coefficients of thrusts near the datasheet coefficients. The predicted coefficients of the thrust are $\sim 4.9\%$ smaller than what the datasheet gives (Table A.1)

$$C_{F_{T_{PSO}}} = \begin{bmatrix} 0.0102 \\ 0.0102 \\ 0.0102 \\ 0.0102 \end{bmatrix} \tag{5.14}$$

The calculated $C_{F_{T_i}}$ are estimated by PSO to be equal for this case, which is the expected result due to the same model of motors and propellers used on the drone. Because the values are within the range of 10%, they are considered to be correct and more accurate than the datasheet values.

As shown in Figure 5.2, the population's best cost is converged to zero over time after the algorithm runs for 35 iterations. This convergence shows that the algorithm works as expected.

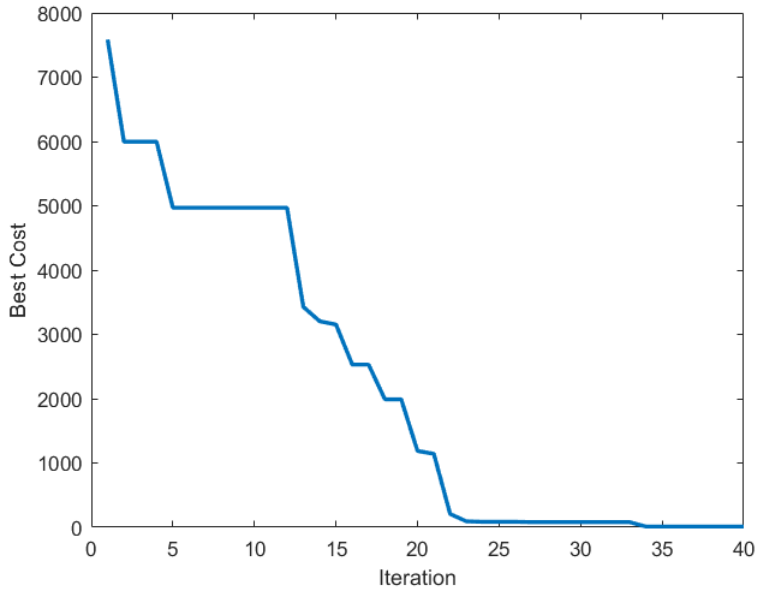

Figure 5.2: Convergence of PSO cost function to zero

Figure 5.3 shows the altitude of the drone in the simulation flights versus the real flight. As seen, the model behaviour when the PSO algorithm is used to achieve the more accurate thrust coefficients is closer to the actual flight. This shows that this mathematical model has a better performance compared to the model that uses the datasheet coefficients. This also shows

that the system identification method proposed in this section can accurately identify the dynamics of the system by only using the flight data.

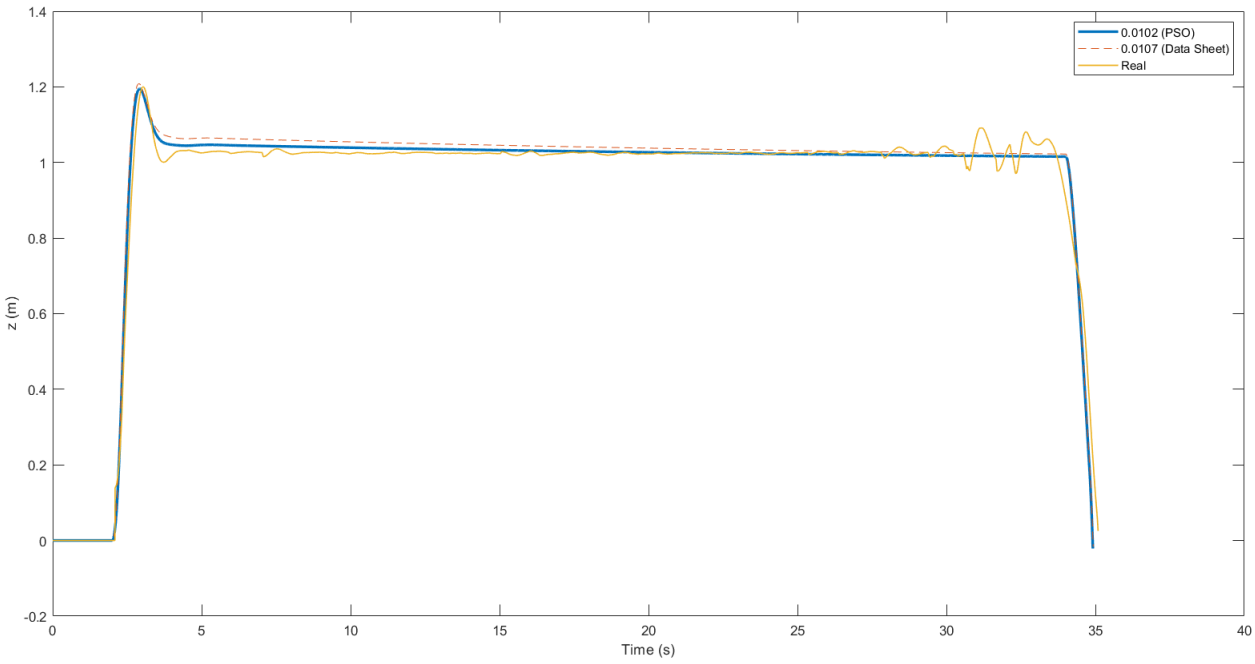

Figure 5.3: The performance of the mathematical model when PSO is used to estimate the thrust coefficients.

Chapter 6

Software

Experimental prototyping of embedded software for real hardware is challenging, and it requires multidisciplinary low-level programming. In other words, researchers are unable to reuse the same code for different hardware since traditional embedded programming practices make it time-consuming and complicated, which inevitably prevents creative investigations in different algorithms.

Since the rapid implementation of new technologies is one of the main goals of this testbed, a novel method is required to reduce the time it takes to implement these technologies on real hardware. This research proposes an embedded programming method that uses Matlab Coder and Embedded Coder. This makes it possible to develop software and perform simulations while simultaneously incorporating it into hardware. The algorithms used in the onboard flight software followed specific guidelines that tailor the high-level Matlab code for automatic C/C++ code generation.

This method was applied in the custom software written for performing

this research. This includes the software for the feed-forward controllers, Vicon Camera integration, the closed-loop controller, command handling, take-off logic, landing controllers, and crash detector. Additionally, the on-board software was integrated with a Matlab model to analyze the system identification and attempt to implement new control algorithms for the drone.

With the ability to implement and simulate the same algorithm on hardware and software, it was possible to conduct this research flexibly. By contrast, using off-the-shelf hardware with built-in software usually implies several restrictions on the flexibility of the hardware and software modifications.

By applying the proposed auto-code-generation method, the algorithms can be developed and tested in a simulation environment. After that, the algorithms are deployed to real hardware without rewriting the code. This method is not limited to the case study of this research, and it has a promising potential to be used for other software applications.

[tab:Traditional-embedded-programming] summarizes the differences between the traditional way of embedded programming versus the proposed method.

Proposed Embedded Programming
Program in a high-level language like Matlab and generate C/C++ code automatically
Fast code prototyping (using Matlab syntax, functions, and toolboxes)
Experimental investigations in algorithms (using Matlab simulation, post-run analysis, and plotting features)
Reusable the code for different hardware or missions
Run and debug the program on a personal computer or the target processor
Independent of hardware
Automatic code replacement to get multithreading and parallelized optimized code

Traditional Embedded Programming
Program in a low-level language like C/C++
Time-consuming, high cost of labour
Hard to do experiments in algorithms
Hard to reuse the code
Run and debug the program only on the target processor
Dependent on hardware
Hard to write parallelized and multithreading optimized code

Table 6.1: Traditional embedded programming vs auto-code generation

Figure 6.1 shows the procedure of the proposed embedded programming. The code prototype employs high-level syntax, functions, and toolboxes available in Matlab. Then the code is tested using Matlab simulation, post-run analysis, and plotting features. Finally, using Matlab coder and Embedded coder, low-level C/C++ code is generated, which is optimized for the specific hardware used in the drone.

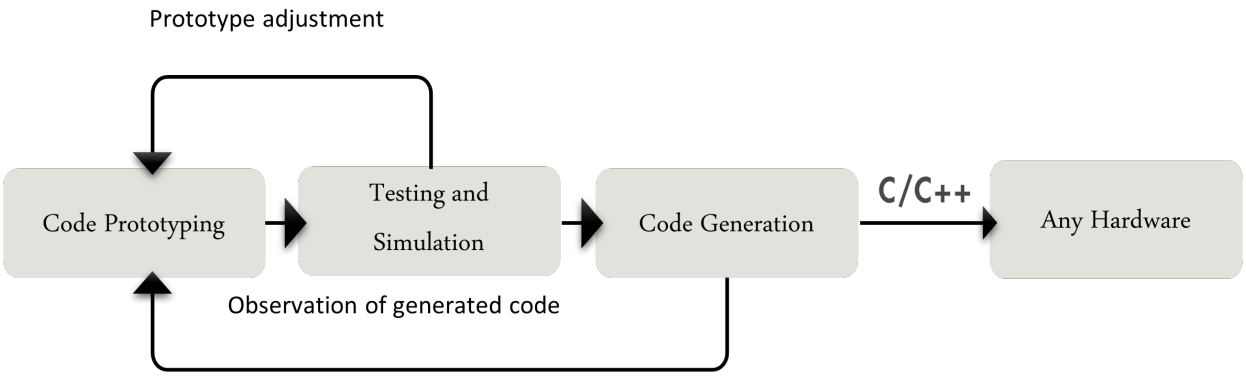

Figure 6.1: Auto code generation procedure

6.1 Onboard software architecture

The architecture of the onboard software is shown in Figure 6.2. Each of the components is described in the following subsections.

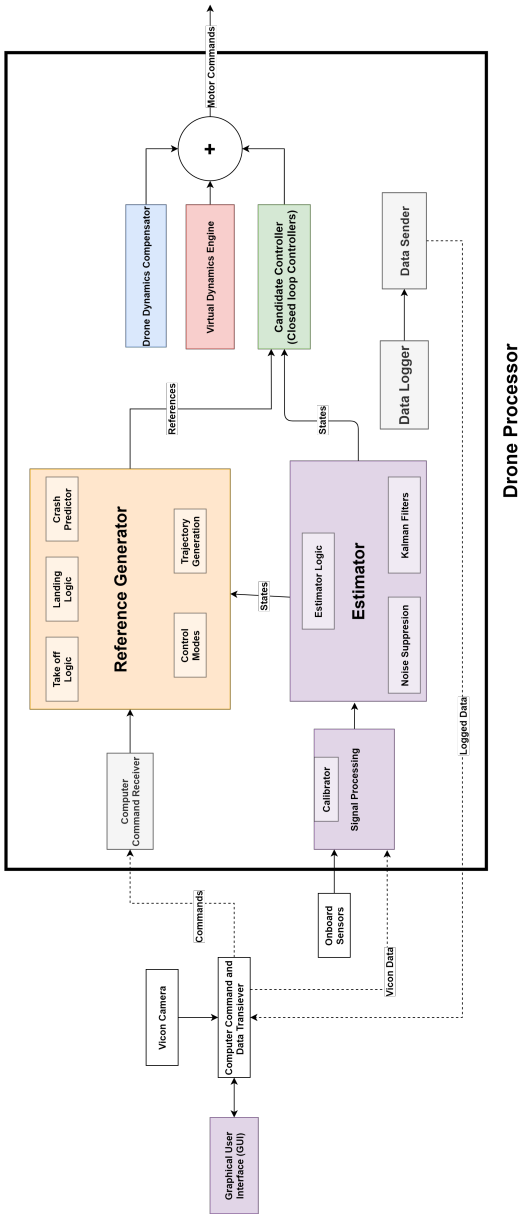

Figure 6.2: Onboard software architecture

The drone processor is the onboard computer responsible for real-time flight control, receiving and processing the ground commands, receiving and processing the Vicon data, and sending the flight log data to the ground station.

6.1.1 Graphical User Interface (GUI)

The graphical user interface, which is shown in Figure 6.3, is an interface that allows the user to:

- specify the flight mode. The flight mode is selectable between feed-forward/closed-loop autonomous control, autonomous closed-loop control, and manual closed-loop control. The feed-forward/closed-loop, autonomous control mode, is the primary mode used in this research. The autonomous closed-loop control mode is used to collect the flight data that is needed for system identification. The manual mode allows manually commanding the drone to take off, land, etc.

- specify the trajectory. If the flight is autonomous, the desired trajectory (the desired set points and their timing) can be specified in the GUI.

- view the flight log. This component receives the flight log and uses various plotting features to process and view the logged data in a human-readable format. This component generates the plots used throughout this research.

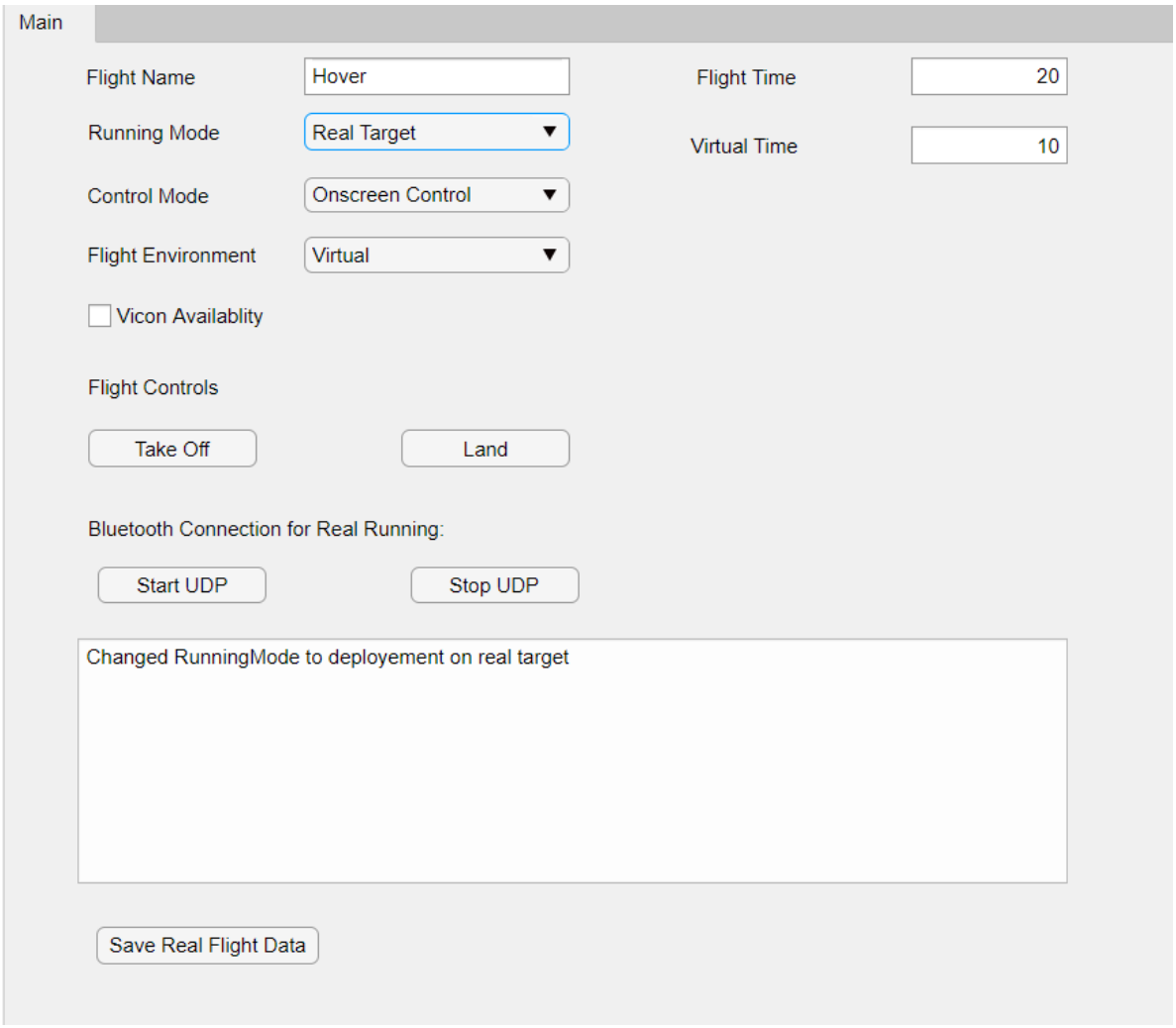

Figure 6.3: The flight control graphical user interface

6.1.2 Computer command and data transceiver

This component is responsible for:

- transferring the commands given by the user to the onboard processor.

These commands include the flight mode, the desired trajectory, etc.

- transferring the Vicon data to the onboard processor.
- receiving the flight log from the onboard processor at the ground station. This log is processed and viewed in the GUI.

These transformations happen over a high-speed wireless network. The real-time data, such as Vicon data, are transferred using the UDP protocol.

6.1.3 Onboard sensors

The onboard sensors, such as an accelerometer, gyroscope, ultrasound altimeter, and camera, measure the drone's accelerations, angular velocities, and position. The measured data is then sent to the onboard signal processor.

6.1.4 Signal processor

The onboard signal processor applies various processes to prepare the measured data received from the onboard sensors and Vicon. The resulting data is then sent to the onboard estimator. These processes include calibration, scaling, etc.

6.1.5 Estimator

The onboard estimator takes the measured data from the sensors and produces the state vector needed for the controller. The onboard estimator has:

- internal logic that decides between the various sources of data such as Vicon and altimeter. It is also responsible for discarding outliers if one

of the sensors measures inaccurate data out of the acceptable range for that sensor.

- noise suppression and Kalman filters remove the noise from the data and then produce a single state vector used in the controllers.

6.1.6 Computer command receiver

This component receives the desired trajectory from the ground station. Depending on the flight mode, this can be a fixed trajectory received at the beginning of the flight or a changing trajectory sent to the drone in real-time.

6.1.7 Reference generator

The reference generator has various parts to produce appropriate desired states that are called reference states for the controllers:

- Control modes and trajectory generator: as discussed in 6.1.1, the user can select various flight modes. This section of the reference generator produces the desired states based on the selected flight mode and the given trajectory.

- Take-off logic: this section decides when the drone should take off

- Landing logic: at the end of the flight, the onboard landing logic described in 3.3.4 reduces the altitude linearly. Accordingly, the closed-loop controllers adjust motor commands based on the changing altitude

generated by the landing logic. Once the drone is near the ground, the motors stop rotating.

- Crash predictor: this part processes the states of the drone to predict and detect crashes. Based on various criteria, when the states are out of the acceptable range predefined in the crash predictor, the crash predictor commands the motors to stop rotating.

6.1.8 Data logger

This part of the onboard processor is responsible for collecting and log the data that must be sent to the ground station. The logged data can be any desired signal. By default, the data logger logs the states vector, the reference states, the motor commands coming from the closed-loop controller, and the motor commands generated by the feed-forward controller.

6.1.9 Data sender

This component sends the data that the data logger logs to the ground so it can be viewed on the graphical user interface that was discussed in section 6.1.1.

6.1.10 Drone dynamics commentator and virtual dynamics engine

The drone dynamics compensator provides sufficient abstraction to enable independent drone control and eliminates in-flight drone dynamics. This involves removing the effects of gravity. Using the virtual dynamics engine,

virtual environments can be emulated, including a different gravity level than the host environment. This section is implemented based on the feed-forward controller that was discussed in Chapter 4.

6.1.11 Candidate controller

This part enables testing any new control algorithms for spacecraft or drones by controlling the drone's motion. These new control algorithms can be state-of-the-art control algorithms that have never been used on-orbit or on the drone. This implementation of this part is based on the closed-loop controller discussed in Chapter 3.

6.2 Programming for code generation

Using the correct data type is crucial for generating efficient code, which is an essential factor for the real-time software of the drone, which runs on an onboard processor with limited available resources.

One of the notable differences between Matlab language and statically compiled languages such as C/C++ is that in Matlab, the type and size of the variables are not declared statically. As a result, the type and size assigned to a variable can change during the program's execution. It is the responsibility of the interpreter and the Just In Time (JIT) compiler to detect the types and sizes during run-time. To avoid these downsides and write Matlab code to be suitable for code generation, the following guidelines were considered. These made it it was possible to implement the algorithms described in this research efficiently.

- **Numbers:** in Matlab, the numbers have double types by default. If double numbers are combined with integer number types in the operation, Matlab implicitly demotes the double types to the lower type. These implicit conversions can result in a loss of precision in the calculations. To prevent such errors, double numbers should be preferred over integer numbers. If there is a guarantee that the integrity of the integer types is preserved, their usage can result in higher performance.

```
n1=5*2.3                    % n1 will be 11.5
n2=uint64(5)*2.3            % n2 will be uint64(11)
n3=uint64(5)*uint64(2)      % n3 will be uint64(10)
```

- **Static Arrays:** static arrays are fixed-size arrays that have a known type at compile-time. To declare such arrays, the following syntax can be used:

```
a=0;                        % 0-dimensional double
b=zeros(2,1);               % 1-dimensional array double
c=zeros(2,4);               % 2-dimensional array double
d=zeros(2,4,3);             % 3-dimensional array double
h=[5,4,3;2,6,7];            % 2-dimensional double with known
    initial value
e=int32(0);                 % 0-dimensional int32
f=zeros(2,1,'uint32');      % 1-dimensional uint32
g=5*ones(5,3,'int8');       % 2-dimensional int8 initialized to
    5
```

- **Static Structures:** statically-allocated structures should be defined using the following syntax. This ensures that the type and size of all the fields are known at compile-time.

```
% 0 dimensional structure
struct1=struct('field1',0,'field2',[0,0],'field3',zeros
    (3,3),'field4','string');

% 1 dimensional structure
dim1=5;
struct2=repmat(struct1,dim1)

% 2 dimensional structure
dim2=3;
struct3=repmat(struct1,dim1,dim2);
```

After initialization, the fields of the structure can be accessed, and their value can change.

```
% changing the field value
struct1.field1=54;
% accessing fields of the 1-dimensional struct
struct2(3).field1=31;
% accessing fields of the 2-dimensional struct
struct3(2,2).field4='hello!'
% possible due to the struct initialization
struct1.field2=[5*struct1.field1,3];
```

A limitation in using multidimensional structures is that the size of values

in the same fields in different structure elements should be the same. For example, the following code raises a compile-time error.

```
% different sizes of val1 in different struct elements
struct2(1).field1=0;
struct2(2).field1=[0;0];
```

To fix this issue, one can make the size of all values the same by using dummy values if the excess overhead is not a concern. Another alternative is to use static cells instead of multidimensional structures.

- **Static Cells:** all the cell array elements on all execution paths should be assigned for statically allocated cells before using or passing to a function. Because there is no native equivalent type for cells in C/C++, static structures are preferred over cells.

```
% cell definition
k=cell(2,1);
% initialize the struct
struct2=struct('field1',0,'field2',zeros(3,3));
k{1,1}=struct2;
k{2,1}=struct2;
% accessing the inner structure fields
k{1,1}.field1=0;
```

- **Dynamic Arrays:** if the size of an array changes dynamically, the range of the size variation should be declared at compile-time to make the generated code efficient.

```
% 1-dimensional array with the 1st dimension being variable
   -size with 5 as the upper bound of the size
coder.varsize('a',[5],[1])

% 2-dimensional array with the 1st dimension having a fixed
    size of 5 and the 2nd dimension being variable-size
   with 10 as upper bound
coder.varsize('b',[5,10],[0,1])

% 2-dimensional array with both dimensions being variable-
   size with 5 and 10 as upper bounds
coder.varsize('c',[5,10],[1,1])

% 2-dimensional array with both dimensions being fixed to 5
    and 10
% this is the same as a static array
coder.varsize('d',[5,10],[0,0])
```

- **Dynamic Structures:** to declare that a field of a structure is dynamically allocated, the following syntax should be used.

```
myStruct=struct('field1',0,'field2',zeros(3,3));
% the 2nd dimension of the field2 can vary up to 5, and the
    size of the 1st dimension is fixed at 3.
coder.varsize('myStruct.field2',[3,5],[0,1]);
```

6.3 Benchmarking auto-code generation

To evaluate the integrity and readability of the generated code, a sample code generated from the attitude controller shown in Figure 6.4 is as follows. As seen, the generated code is human-readable and preserves the logic of the original code correctly.

```
if (localB->ReferenceValueCmd.takeoff_flag) {
  localB->time = localP->takeoff_gain_Gain * localP->
     weight_Value;
} else {
  localB->time = (localP->P_z_Gain * localB->err_z +
                  localDW->DiscreteTimeIntegrator_DSTATE_p)
                - localP->D_z_Gain * localB->states.dz;
}
localB->Sum = localP->weight_Value + localB->time;
```

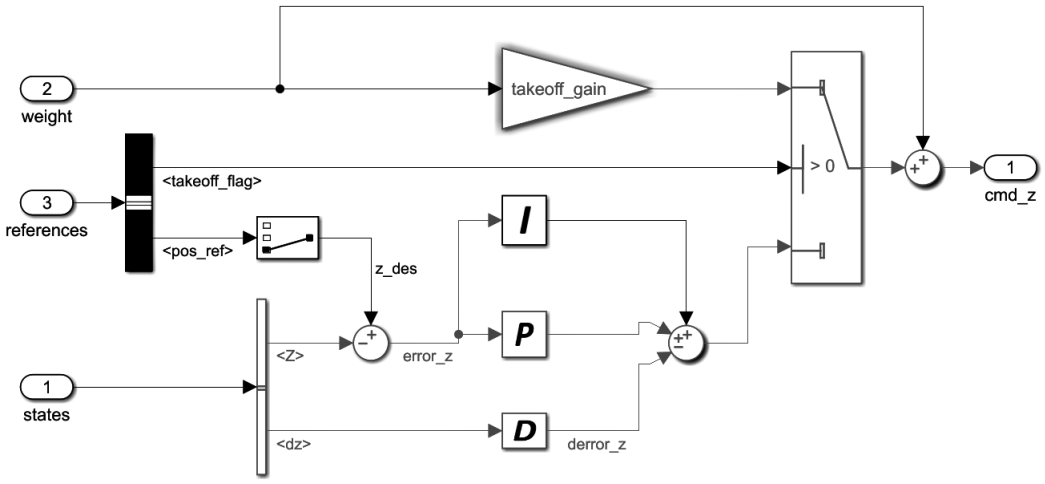

Figure 6.4: A sample Simulink file used to demonstrate code-generation

To further evaluate the performance of the generated code, a set of algorithms were developed in two separate groups. The first group was developed by manually writing low-level C code. The second group was written in Matlab, and the Embedded Coder was used to generate low-level C code. Then the performance of these two groups was collected and compared for benchmarking.

As the primary benchmark test for this case study, the generated C code performance is tested on a Zybo Z7-20 [59], which is a development board based on the Xilinx Zynq Z-7020 system on the chip (SoC) architecture [60]. It has a dual-core ARM Cortex-A9 processor and an FPGA equivalent to an Artix-7 FPGA. For the real-time operating system of the ARM cortex processor, Linux Linaro was used.

The algorithm selected for this benchmark is solving a linear system of equations and then calculating the inverse of a matrix. The matrices are

randomly generated double numbers between 0 and 1.

The structure of the Matlab code is as follows:

```
function [x, invA] = mainF()
        [A, B, C] = dataLoader();
        [x, invA] = algorithm(A, B, C);
end
```

The `dataLoader` function is

```
function [A, B, C] = dataLoader()
        coder.inline('never');

        A = rand(200, 200);
        B = rand(200, 1);
        C = rand(100, 100);
end
```

The `algorithm` function is

```
function [x, invA, mulAb] = algorithm(A, b, C)
        coder.inline('never');

        x = A \ b;
        invA = inv(C);
end
```

The handwritten C code is given in Appendix C, in which the Gaussian elimination algorithm was used to solve the linear system, and the LU decomposition algorithm was used to calculate the inverse of the given matrix.

The average performance for the auto-generated code is

```
CPU time (secs): user=0.2645; system=0.0000; real=0.2644
```

The average performance for the hand-written code is

```
CPU time (secs): user=0.0704; system=0.0000; real=0.0703
```

The above benchmarks show that the meticulously hand-optimized code has better performance than the auto-generated code. However, when the error-prone and time-consuming development of low-level C code is taken into account, the high-level Matlab code written in 10 lines of code becomes a very compelling choice in this research.

The auto-generated C code is readable, making it possible to review or edit the resulting code if desirable optionally. For example, a part of the algorithm generated for the above linear system solving the problem, with the preserving dimension option disabled, is as follows:

```
#include <string.h>
#include "mainF.h"
#include "mldivide.h"
#include "xtrsm.h"
#include "xgetrf.h"

void mldivide(const double A[40000], double B[200])
{
  static double b_A[40000];
  int ipiv[200];
  int info;
  double temp;
  memcpy(&b_A[0], &A[0], 40000U * sizeof(double));
```

```
  xgetrf(b_A, ipiv, &info);
  for (info = 0; info < 199; info++) {
    if (ipiv[info] != info + 1) {
      temp = B[info];
      B[info] = B[ipiv[info] - 1];
      B[ipiv[info] - 1] = temp;
    }
  }
  xtrsm(b_A, B);
  b_xtrsm(b_A, B);
}
```

6.4 Conclusion

In this chapter, an embedded programming method was introduced that uses the Matlab Coder and Embedded Coder in a structured and organized manner to generate efficient C/C++ code automatically. The method is ideal for implementing new algorithms due to the reduced development effort. This significantly contributes to the practicality and accessibility of new technologies by making it possible to test and implement them.

Chapter 7

Results and Conclusion

In this chapter, the results for the system identification method and feed-forward control are presented. This chapter presents the results related to the Parrot Mambo drone, in which its physical properties are given in Section §A. These results can be extended to other types of drones and configurations.

7.1 Feed-forward control results

Based on the experiments conducted in the testbed environment, the transition from the closed-loop control to feed-forward control plays a vital role in the system's stability. The feed-forward command injection gain can have any of the shapes depicted in Figure 7.1.

Due to the complex system dynamics, environmental disturbances, and the system's imperfections, if the feed-forward terms are applied from the beginning of the flight, the drone will fail to take off. Even if they are enabled at one point after that, the drone has reached a stable hover state,

the sudden change in the commands results in instability of the system.

As a result, the feed-forward control commands should be injected into the system incrementally instead of using a constant gain. In the following subsections, the results for a linear feed-forward injection gain and a sigmoid feed-forward injection gains are presented.

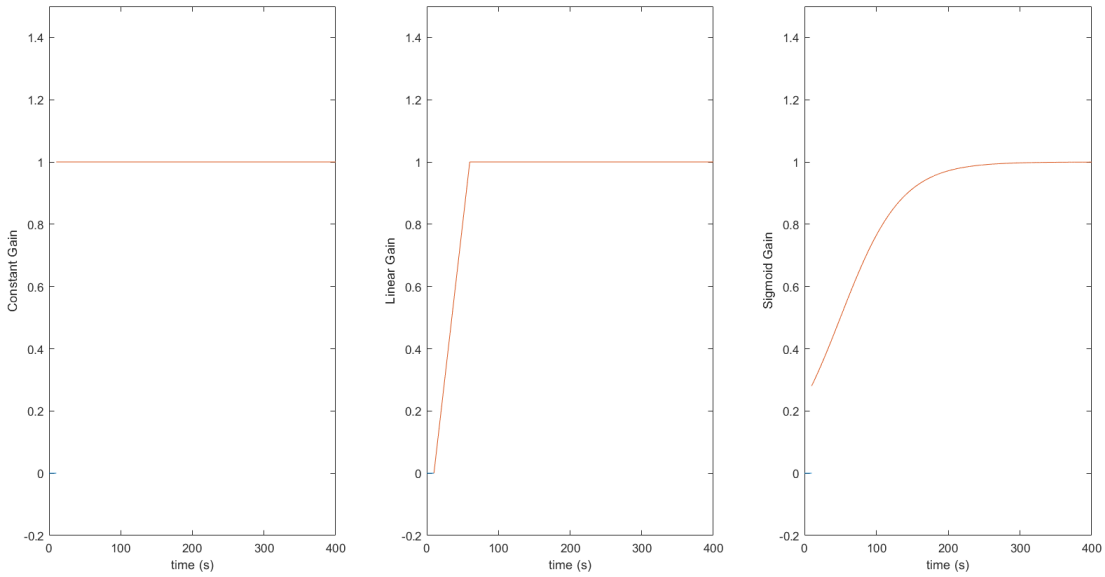

Figure 7.1: The feed-forward injection gain curves

7.1.1 Linear feed-forward injection gain

In this section, the feed-forward controller command is injected into the system using a linear gain, which means when the drone has reached a stable

hover state, the feed-forward control commands are increased with a linear gain until they reach the planned value.

A specific time at which the feed-forward controller starts to work is called $T_{feed-forward}$. This time needs to be chosen so that it is after the drone reaches a stable hover state. For the configuration of this experiment, $T_{feed-forward}$ is chosen to be $10s$.

This gain is a function of time that is passed from $T_{feed-forward}$. This linear gain is formulated as:

$$gain_{feed-forward} = K_{slope} \frac{t - T_{feed-forward}}{T_{feed-forward}} \tag{7.1}$$

K_{slope} is a constant that specifies the rate at which the feed-forward controller command is increased. Based on the experiments, it is found that a K_{slope} around 0.2 is a proper value for the configuration of our drone. This rate allows the closed-loop controllers to adapt their outputs smoothly.

After the $gain_{feed-forward}$ reaches $gain_{feed-forward_{max}}$, the value of the gain will saturate. This prevents the gain from increasing without bounds. This maximum value is used to emulate an environment with reduced gravity. For example, if $gain_{feed-forward_{max}}$ is set to 0.4, an environment with 36% reduced gravity is emulated[1].

To get zero-gravity, $gain_{feed-forward_{max}}$ should be set to 1. This allows the feed-forward controllers to reach their highest values. For the closed-loop controllers chosen for this research, reaching pure zero-gravity is not practical in a real-world situation. The reason is that the controllers used in this research are designed for $1g$ environments, and they are not robust in

[1] $(1 - 0.4)^2 = 0.36 = 36\%$

a reduced gravity environment. As a result, the slightest disturbance in the angles will cause the drone to tilt, and the drone becomes unstable.

If a virtual environment with 25% reduced gravity is desired, then $gain_{feed-forward}$ is considered to be 0.5 to achieve such a virtual environment.

$$(1 - 0.5)^2 = 0.25 = 25\% \text{ reduced gravity}$$

The altitude of the drone in this configuration is shown in Figure 7.2. The feed-forward command reaches the planned value at $t = 80\,s$. The average altitude converges to $1\,m$ after the feed-forward control is reaches the planned value.

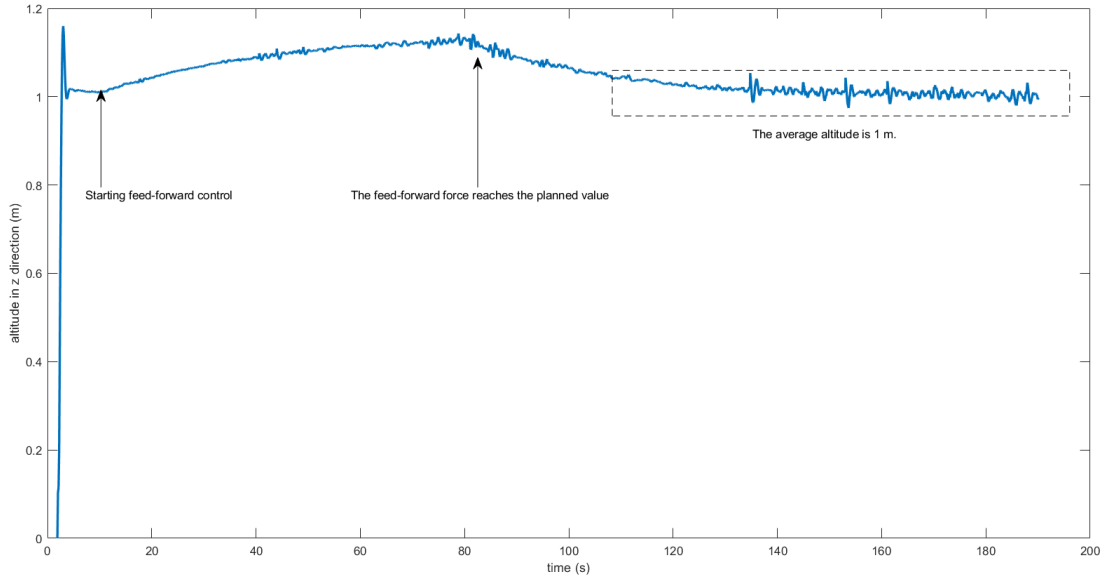

Figure 7.2: The altitude of the drone in a virtual environment with 25% reduced gravity

As seen in Figure 7.3, the average of the total command sent to motor 3 is constant over time. This is because the feed-forward controller and closed-loop controllers are working together to hold the average altitude of the drone constant.

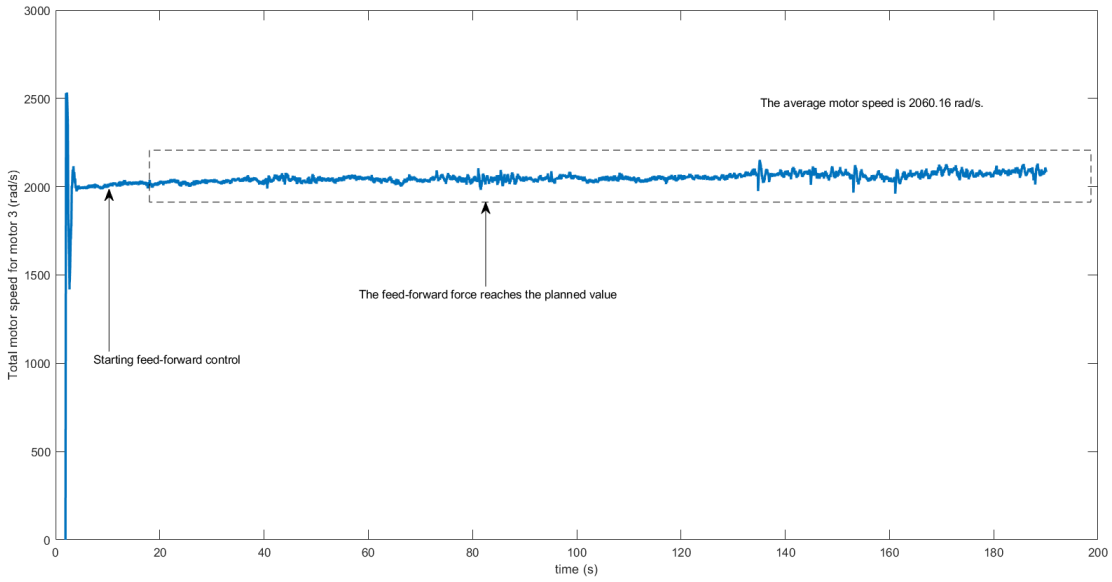

Figure 7.3: The total command sent to motor 3 in a virtual environment with 25% reduced gravity

The Figure 7.4 and Figure 7.5, which are showing the closed-loop commands sent to motors 3 and 4, indicate that the feed-forward controller has caused a reduction in the closed-loop commands. The average closed-loop commands for $t > 80\,s$ are $1305.5\,\frac{rad}{s}$ and $-1373.98\,\frac{rad}{s}$ respectively.

The performance of the feed-forward controller is proportional to the inverse of the average of the closed-loop commands once they reach a stable value. This measure can be formulated as the following error:

$$\text{Feed-forward command error} = \left| \frac{\text{The planned command - The average command}}{\text{The planned command}} \right| \tag{7.2}$$

The planned motor command can be calculated from:

$$\text{The planned command} = gain_{feed-forward_{max}} * \text{The maximum motor speed} \tag{7.3}$$

The planned motor command for this case is

$$0.5 * 2630.7 \, \frac{rad}{s} = 1315.35 \, \frac{rad}{s} \tag{7.4}$$

The feed-forward command error for the 3rd controller when $t > 80\,s$ can be calculated as

$$\frac{1315 - 1305.5}{1315} = 0.007 \tag{7.5}$$

For the 4th controller, this error is

$$\left| \frac{-1315 - (-1373.98)}{-1315} \right| = 0.0448 \tag{7.6}$$

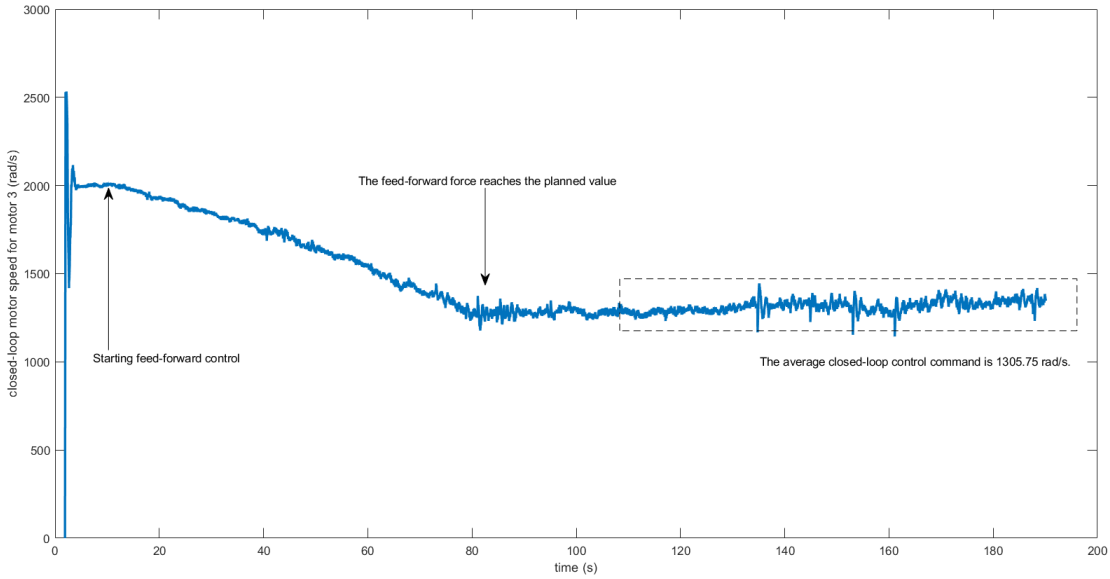

Figure 7.4: The closed-loop command sent to motor 3 in a virtual environment with 25% reduced gravity

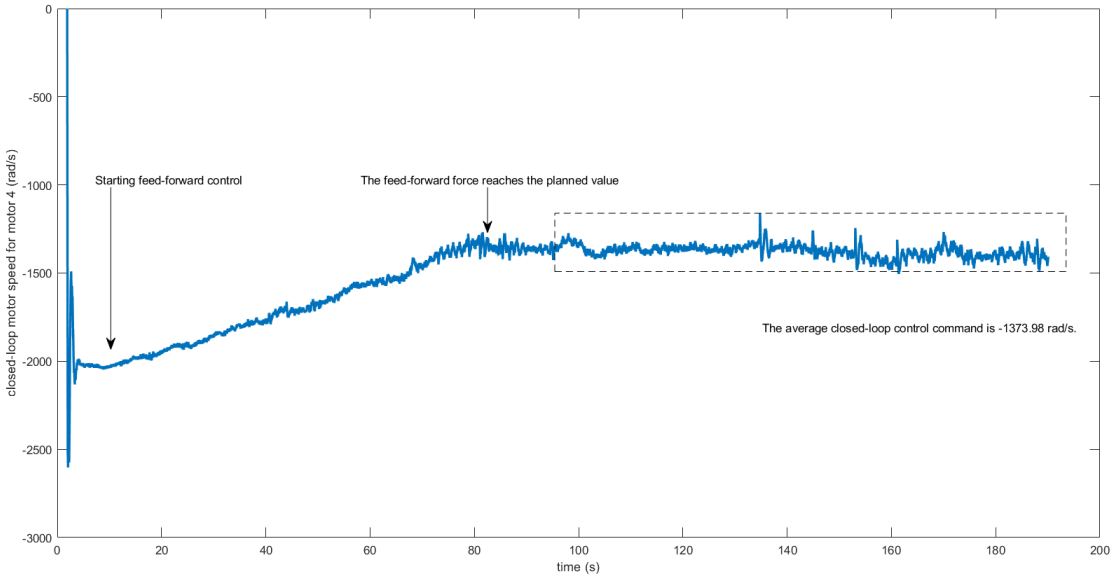

Figure 7.5: The closed-loop command sent to motor 4 in a virtual environment with 25% reduced gravity

When a virtual environment with 64% of reduced gravity is desired, $gain_{feed-forward}$ should be set to 0.8. The altitude of the drone for this environment is shown in Figure 7.6. In this particular case, the disturbance caused when the feed-forward controller reaches the planned value causes an initial increase in the drone's altitude during the reduced gravity period. If the drone continued the flight for an extended amount of time, it had settled back at $1\,m$.

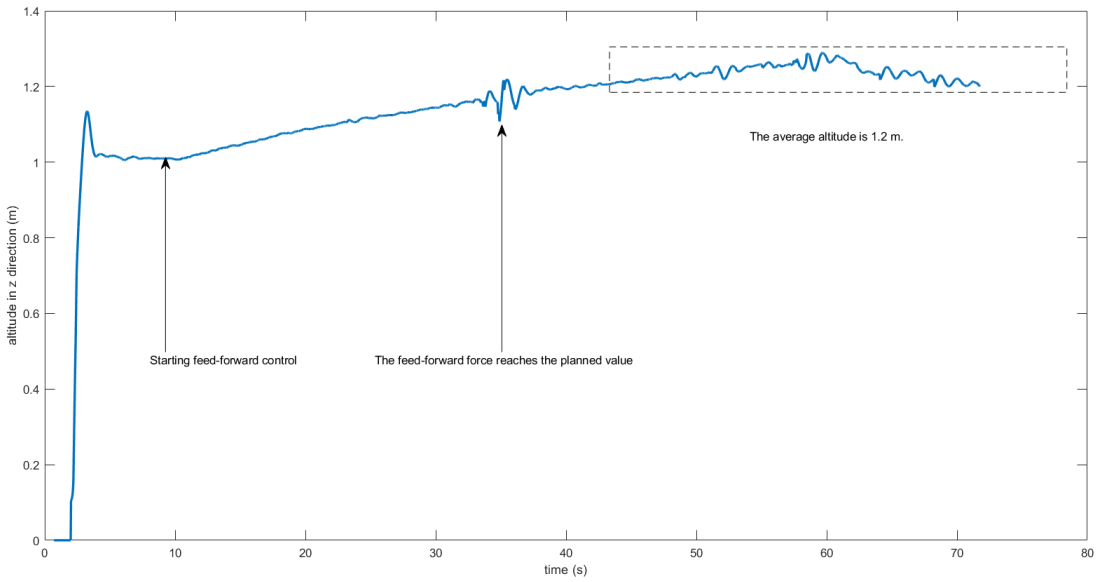

Figure 7.6: The altitude of the drone in a virtual environment with 64% reduced gravity

The closed-loop commands in this environment are plotted in Figure 7.7. As seen, the closed-loop controller is only working at its $\sim$ 36% capacity, which means the feed-forward controller has reduced the gravity by 64%. The average closed-loop commands for $t > 60\,s$ are $596.57\,\frac{rad}{s}$ and $-596.48\,\frac{rad}{s}$ respectively.

The planned motor command for this case is

$$0.2 * 2630.7\,\frac{rad}{s} = 526.14\,\frac{rad}{s} \tag{7.7}$$

The feed-forward command error for the 3rd controller when $t > 60\,s$ can be calculated as

$$\left|\frac{596.57 - 526.14}{526.14}\right| = 0.1338 \tag{7.8}$$

For the 4th controller, this error is

$$\left|\frac{-526.14 - (-596.48)}{-526.14}\right| = 0.1337 \tag{7.9}$$

According to Figure 7.8, the averages of total closed-loop commands are kept constant after the initial take-off period.

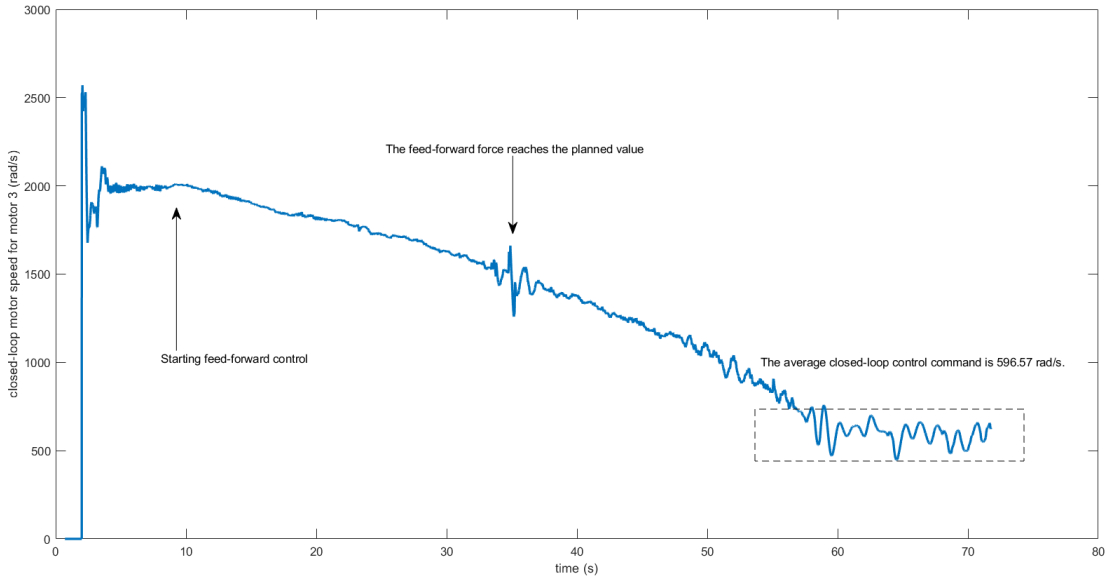

Figure 7.7: The closed-loop command sent to motor 3 in a virtual environment with 64% reduced gravity

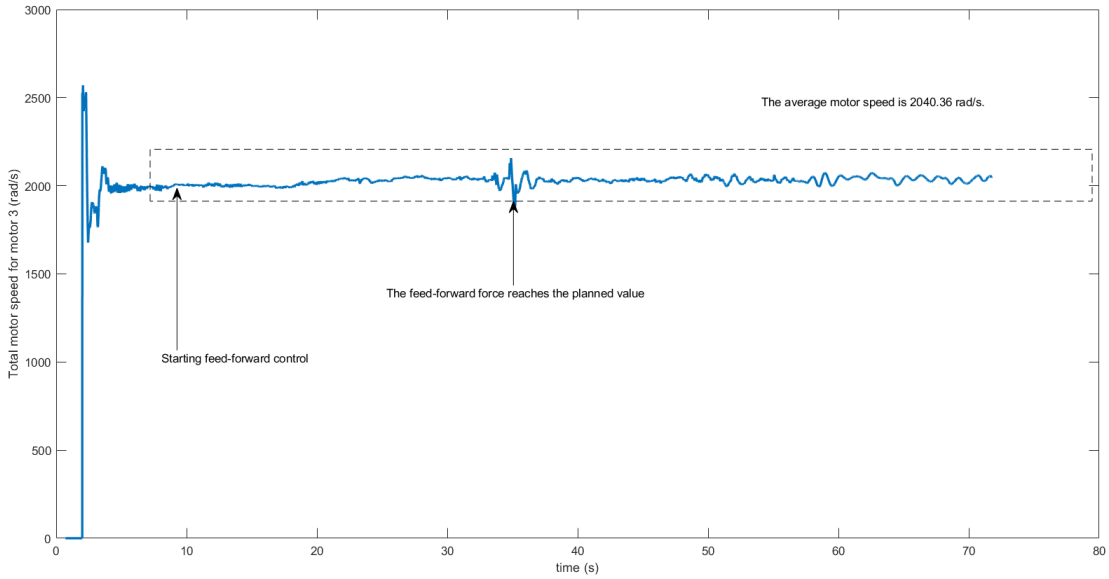

Figure 7.8: The total command sent to motor 3 in a virtual environment with 64% reduced gravity

The importance of accurate dynamics is visible in Figure 7.9. This plot shows the commands coming from the closed-loop controllers over samples. As seen, the more accurate the coefficients of the thrust are, the lower the closed-loop controller commands go. This means the feed-forward controller is doing more work, and thus the environment is nearer to zero gravity. As seen, different values of thrust coefficients with different accuracies are used to show the importance of knowing the correct value. The lowest line is for the case where thrusts' coefficients are identified by PSO, which is the closest to the ideal line. The thin and the dash lines are respectively related to the thrust coefficients with 20% and 50% errors. These two cases are also less stable due to the inaccurate feed-forward control planning caused by the incorrect coefficients, and so a slight disturbance can result in a drone crash.

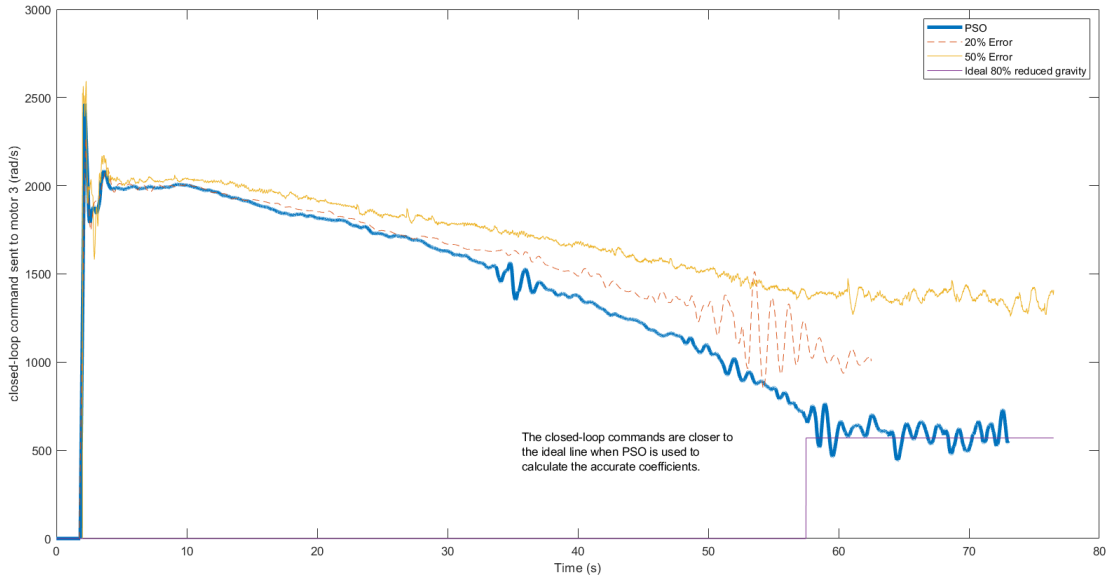

Figure 7.9: The effect of correct coefficients of thrust on a lower closed-loop command in the feed-forward mode

Calculating the feed-forward command error for the three cases in Figure 7.10 gives:

$$\text{Feed-forward command error} = \begin{cases} \frac{605-570}{570} \approx 0.06 & PSO \\ \frac{995-570}{570} \approx 0.75 & 20\%\ Error \\ \frac{605-570}{570} \approx 1.35 & 50\%\ Error \end{cases} \tag{7.10}$$

It is clear that the accurate coefficients result in precise feed-forward control, and so the error for the experiment that used PSO is lower than in the other cases.

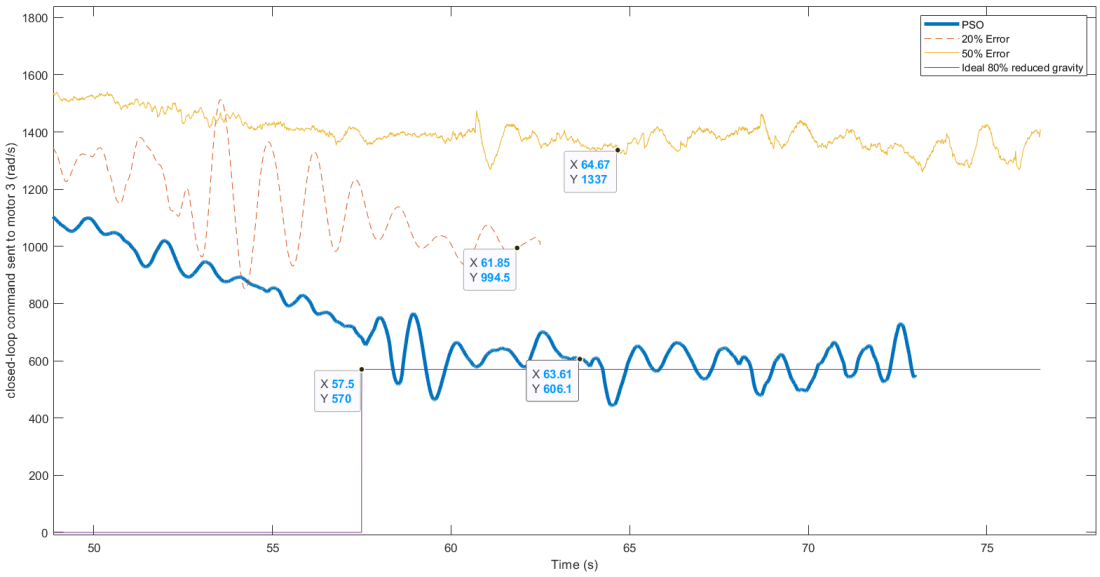

Figure 7.10: The feed-forward control performance measure (zoomed-in)

7.1.2 Sigmoid feed-forward injection gain

To improve the performance of the feed-forward controller, in this section, a sigmoid function is chosen for the injection gain. This sigmoid gain is formulated as:

$$gain_{feedforward} = \left(\frac{K}{(1 + c^{(at+b)})} \right) \tag{7.11}$$

Where K, a, b, c are constant gains that shape the sigmoid function. K specifies the final value that the function will reach by shifting the function vertically. a specifies the slope of the increasing sigmoid curve. If a is negative, the sigmoid function will increase over time and vice versa. b specifies the time when the gain reaches its final value by shifting the function horizontally. Finally, c specifies the exponent by which the function increases.

Similar to the linear gain discussed in the previous subsection, $gain_{feed-forward_{max}}$ is also used for saturation.

After trial and error to get a sigmoid function that gives the best feed-forward control performance for a 49% reduced gravity environment, these gains were selected as in Equation (7.12). The resulting sigmoid function is depicted in Figure 7.11. The smooth increasing of this function does not have the rough edges of the linear injection gain that could result in less stable flight and thus lower feed-forward control performance. Similar to before, $T_{feed-forward}$ is chosen to be $10s$, and so the gain starts from $t = 10s$.

$$\begin{cases} a = 30 \\ b = -0.6 \\ c = 1.04 \\ K = 1 \\ gain_{feed-forward_{max}} = 0.7 \end{cases} \tag{7.12}$$

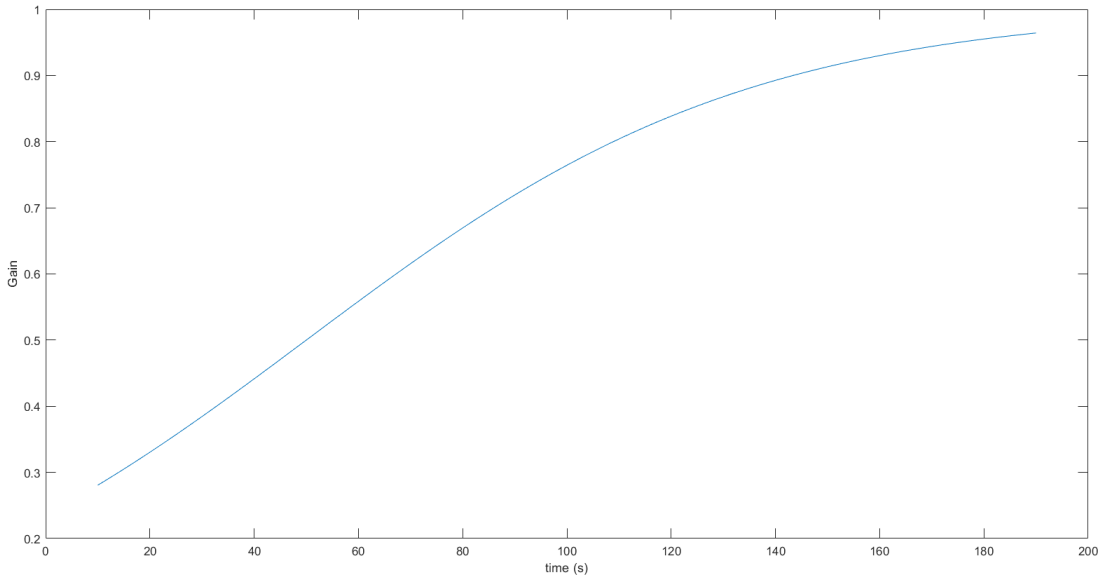

Figure 7.11: The sigmoid feed-forward injection gain over time

The altitude of the drone in this environment is shown in Figure 7.12. As seen after the initial altitude increase that is caused by the initial value of the feed-forward gain, the drone maintains a constant altitude.

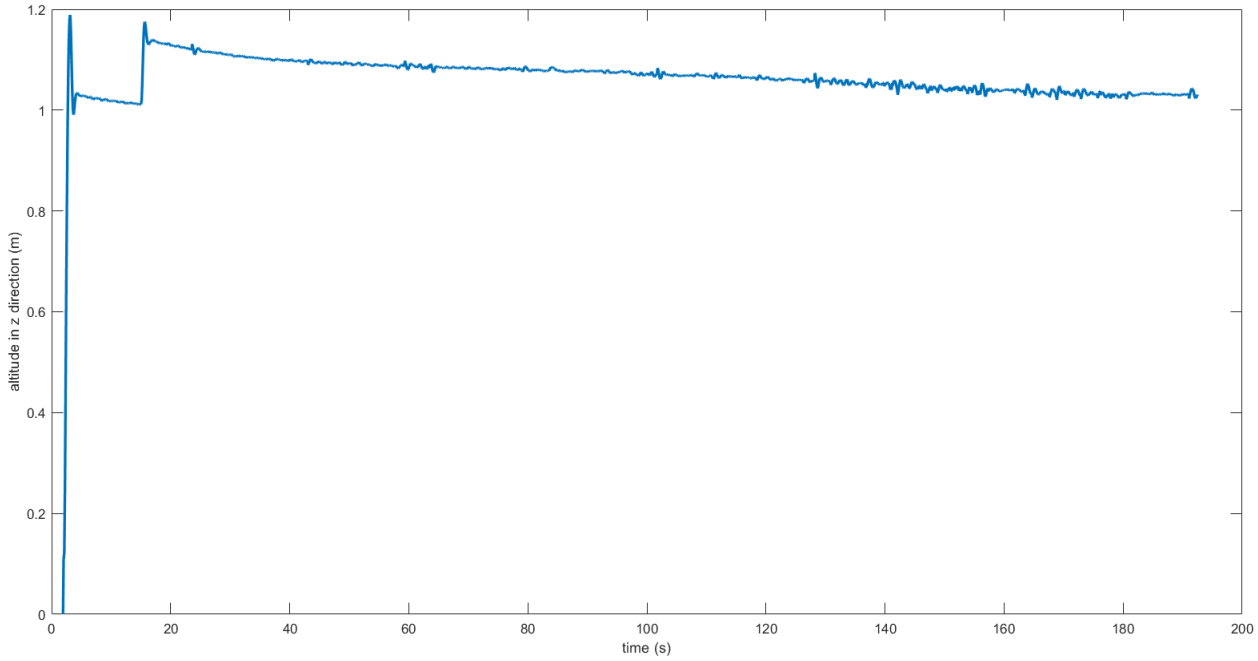

Figure 7.12: The altitude of the drone in a virtual environment with 49% reduced gravity and a sigmoid injection gain

The velocity of the drone in this configuration is shown in Figure 7.13. After the initial take-off state, the velocity is at zero during the flight.

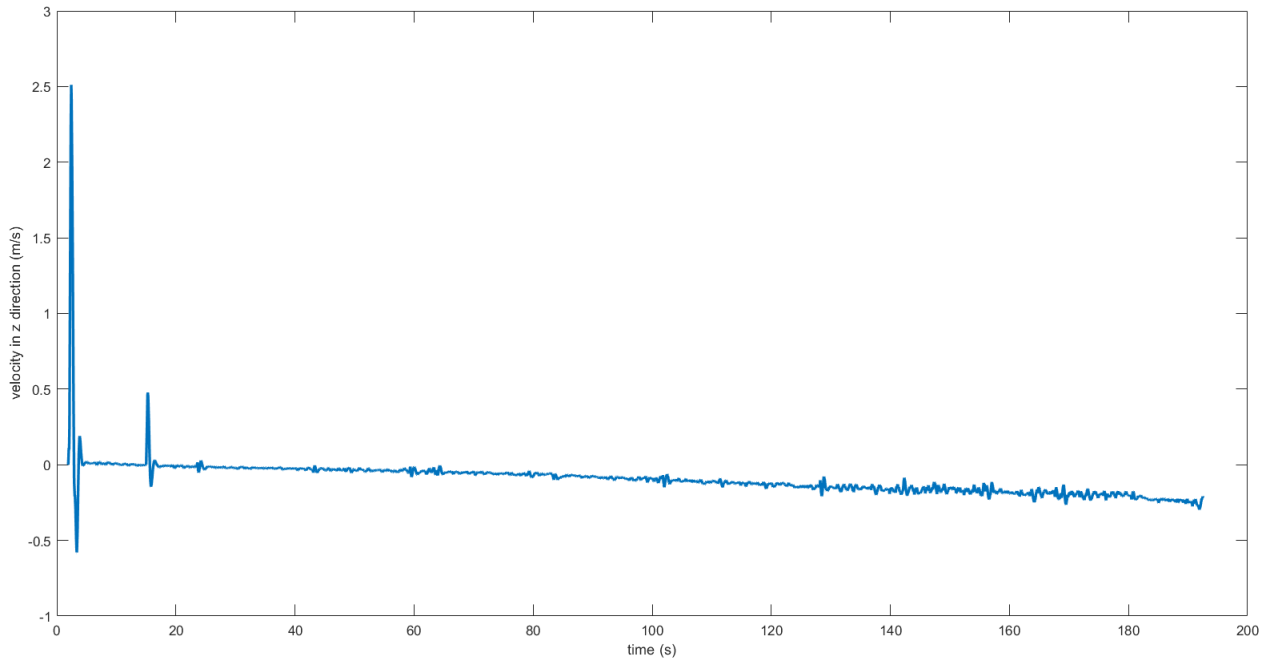

Figure 7.13: The velocity of the drone in a virtual environment with 49% reduced gravity and a sigmoid injection gain

The commands sent to the motors 3 and 4 are shown in Figure 7.14 and Figure 7.15. The average motor command is reduced to 762.903 $\frac{rad}{s}$.

Since the ideal command is $0.3 * 2630.7 = 798.21 \frac{rad}{s}$, the feed-forward error is calculated as

$$\left| \frac{798.21 - 762.903}{798.21} \right| = 0.0329 \tag{7.13}$$

This means that the 49% reduced gravity environment is achieved in this configuration with 3.29% of error.

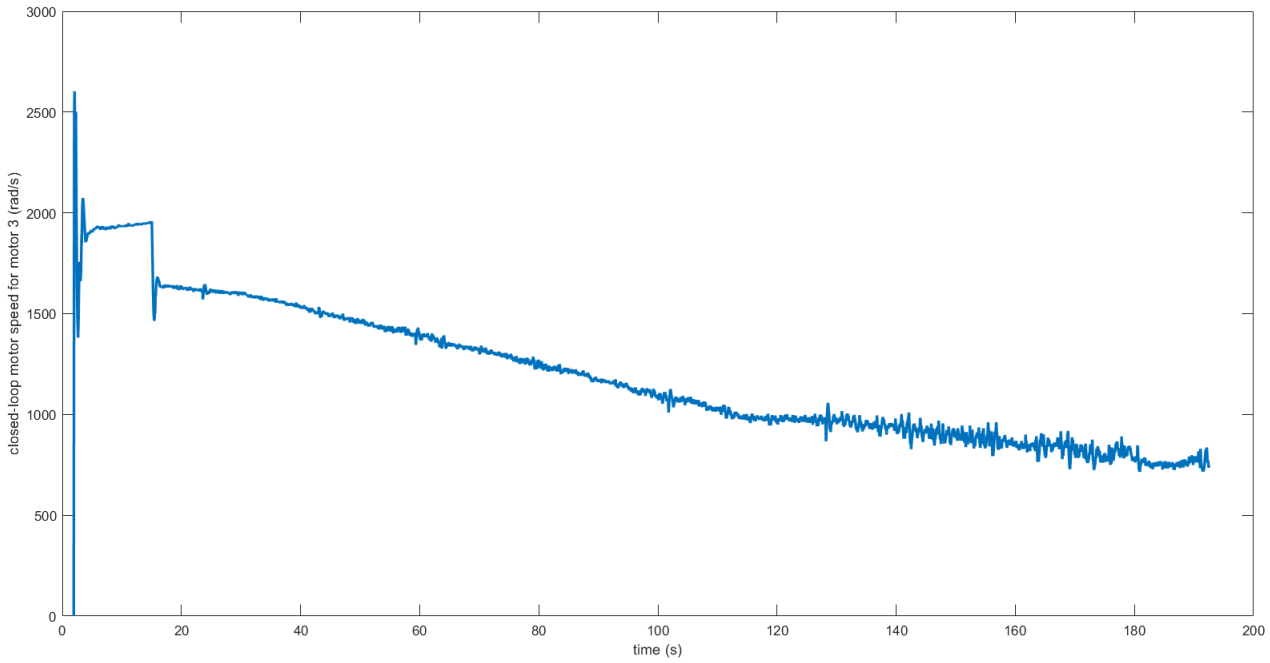

Figure 7.14: The closed-loop command sent to motor 3 in a virtual environment with 49% reduced gravity and a sigmoid injection gain

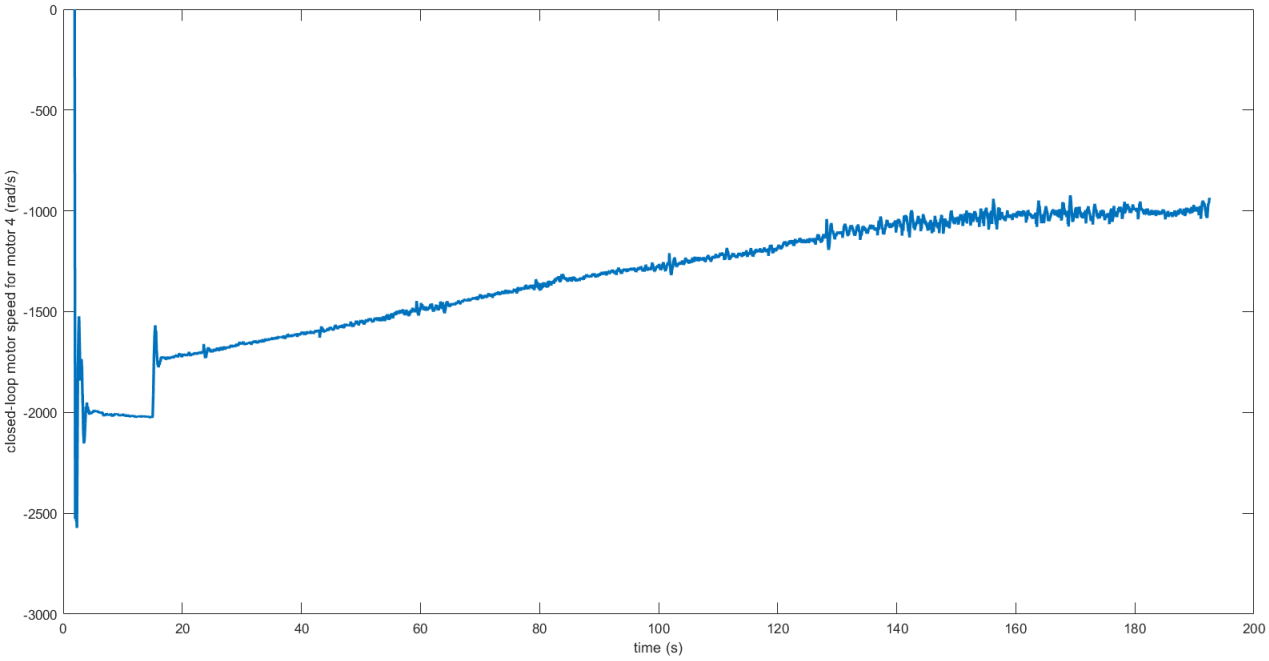

Figure 7.15: The closed-loop command sent to motor 4 in a virtual environment with 49% reduced gravity and a sigmoid injection gain

7.2 Conclusion

In this thesis, a drone testbed was developed as a platform for rapid development, validation, and qualification of novel spacecraft, drone control algorithms, and remote sensing technologies.

By simulating various dynamic environments, the drone testbed facilitates advanced navigation and control research, thereby enabling new technologies in industrial applications. This addresses the first hypothesis, which intended to use drones as a testbed for developing new technologies by emulating virtual dynamics using feed-forward control.

In this study, an approach for determining the dynamics and aerodynamic parameters of the drone using particle swarm optimization (PSO) was introduced.

Utilizing this system identification method, the system's dynamics were identified, which allowed for the emulation of virtual environments such as reduced gravity via a feed-forward controller. This feed-forward controller worked in parallel to the candidate closed-loop controller.

Based on the conducted experiments, it was determined that the performance of the feed-forward controller depended on the injection gain by which the feed-forward controller becomes active.

As discussed, using a sigmoid function achieved better results compared to using a constant gain or a linear gain. In a sigmoid function, there are no sudden changes in the feed-forward control behaviour, and the smooth curvature of the function makes the flight stable even though the commands sent by closed-loop controllers were reduced. This was tested for different configurations, including a 49% reduced gravity.

A mathematical model was developed to simulate drone dynamics and aerodynamics. This model was used later as part of gray box system identification and the onboard flight software of the drone. PSO was used to make the mathematical model match the collected flight data. So, as stated in the second hypothesis, the drone dynamics parameters can be identified with minimal direct measurement using flight data. PSO enabled the estimation of system parameters from the collected data.

As discussed throughout this thesis, accurate knowledge of the drone's dynamics is crucial to get higher performance from the feed-forward controller.

The particle swarm optimization algorithm was needed to find the parameters that describe the drone's dynamics to achieve such accurate knowledge. These parameters made the mathematical model act like the real-world collected data.

The software needed for the onboard computer and the ground station was implemented using auto-code generation. This method reduced the development effort and allowed for more flexibility in the algorithms. "Performant custom flight and ground software can be automatically generated from a high-level language," as stated in the third hypothesis.

7.3 Future work

Given the versatility and the generality of the methods discussed in this work, various future applications and extensions can be considered. In this section, some of them are summarized:

- Because in the system identification proposed in this research, the mathematical model is solved numerically, the dynamics and aerodynamics are not required to be derivable. So, the mathematical model used to identify the dynamics of the model can be extended by more advanced aerodynamics. These models can consider flight situations that are different from those used in this research. For example, in an outdoor flight, the effect of the external wind can change the dynamics of the model.

- The feed-forward control that was designed in this research can be implemented as a companion to different closed-loop. The addition of the feed-forward controller can improve the efficiency and performance of the controlling.

- When fully decoupled feed-forward control is desired, a drone system can be designed that gives six independent degrees of freedom. Such design lifts the limitation of the current drone designs that are underactuated. Having separate motors on the sides of the drone allows for implementing feed-forward controllers that can emulate various horizontal dynamics.

- In parallel to using particle swarm optimization in the system identification method, a black-box model such as an artificial neural network can improve the areas that the gray-box method has not considered.

Appendix A

Parrot Mambo physical properties

In table A.1, all the physical properties of Parrot Mambo is presented. Parrot Mambo was the main drone that was used to collect the final result.

Table A.1: Parrot Mambo physical properties

property	value
mass (m)	$0.07\,kg$
inertia matrix in body fixed frame (I)	$\begin{bmatrix} 0.0000582857 & 0 & 0 \\ 0 & 0.0000716914 & 0 \\ 0 & 0 & 0.0001 \end{bmatrix}$ $kg.m^2$
arm length (l)	$0.0624\,m$
motor number	4
propeller radius (R)	$0.033\,m$
propeller disc area (A_{prop})	$0.0034\,m^2$ Equation (2.33)

air density (ρ)	$1.184\ \frac{kg}{m^3}$
datasheet thrust force coefficient (C_{F_T})	$\begin{bmatrix} 0.0107 \\ 0.0107 \\ 0.0107 \\ 0.0107 \end{bmatrix}$
datasheet drag moment coefficient (C_{M_D})	$\begin{bmatrix} 0.7826 \\ 0.7826 \\ 0.7826 \\ 0.7826 \end{bmatrix} 10^{-3}$
command to motor speed square gain	13840.8
maximum motor speed	$2630.7\ \frac{rad}{s}$[1]
minimum motor speed	$0\ \frac{rad}{s}$

[1]This is equal to the square root of the multiplication of the maximum possible command (500) and the command to motor speed square gain.

$$\sqrt{13840.8 * 500} = 2630.7$$

Appendix B

The experimental setup

Figure B.1 shows the drone testbed. This testbed includes a copter-style drone flying in an enclosed space that is equipped with Vicon motion capture cameras. The computer interface allows for measuring, testing, and analyzing data.

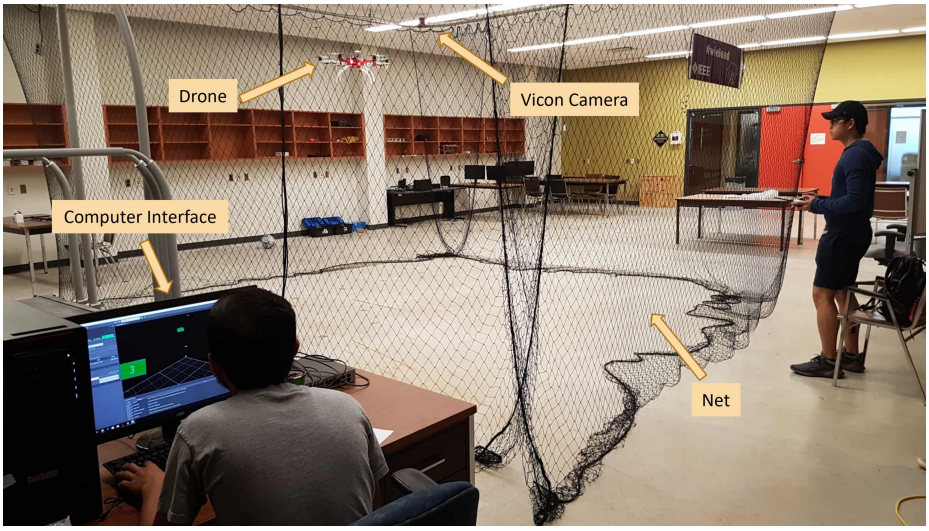

Figure B.1: The drone testbed

This following list gives the equipment and resources that were used in this testbed.

- The Parrot Mambo quadcopter that is shown in Figure B.2:
  - The processor: ARM Cortex A9
  - The onboard sensors: accelerometer, gyroscope, ultrasound altimeter, and camera
  - The communication network: the ground station commands and Vicon data are sent over Bluetooth using the UDP protocol

Figure B.2: Parrot Mambo

- The Vicon cameras, which are shown in Figure B.1 and Figure B.3.

- The DJI F550 hexacopter that is shown in Figure B.4.
  - Pixhawk autopilot

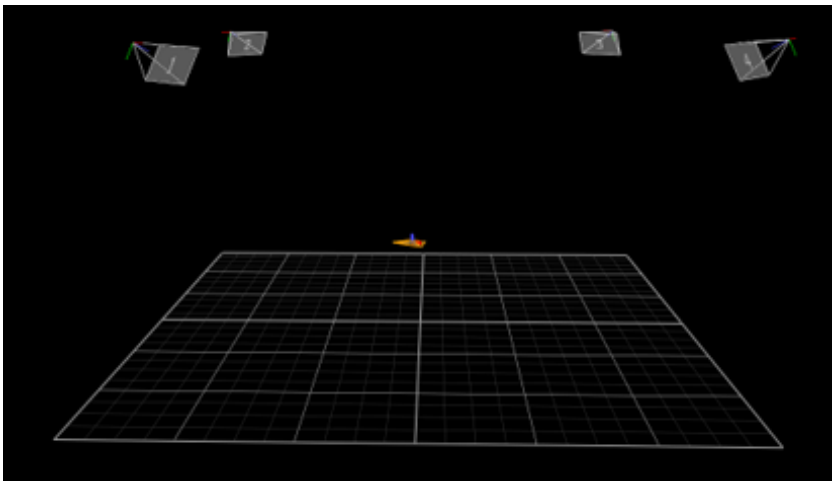

Figure B.3: Vicon Cameras

  - Onboard compass, accelerometer, and gyroscope
  - The Custom module to receive Vicon data
  - The ARM cortex microcontroller
  - The wireless radio
  - The Vicon SDK code to send fake GPS data to the drone

- The ground station computer that is shown in Figure B.1.

Figure B.4: DJI F550 hexacopter

Appendix C

Handwritten C code used for benchmarking

The following is the handwritten C code that implements the Gaussian elimination algorithm for solving a linear system, and LU decomposition to calculate the inverse. This algorithm was compared to the auto-generated code in Section §6.3.

```
#include <stdio.h>
#include <stdlib.h>

/** Generate random double numbers between 0 and 1 */
double double_random()
{
    return (double)rand() / RAND_MAX;
}
```

```
/** Generate a random array with length of `len` */
void random_array(int len, double arr[len])
{
    for (int ielm = 0; ielm < len; ++ielm)
    {
        arr[ielm] = double_random();
    }
}

/** Generate a random matrix with row length of `row_len`
   and column length of
 * `column_len` */
void random_matrix(int row_len, int column_len, double mat[
   row_len][column_len])
{
    for (int irow = 0; irow < row_len; ++irow)
    {
        for (int icolumn = 0; icolumn < column_len; ++
           icolumn)
        {
            mat[irow][icolumn] = double_random();
        }
    }
}

/** print the content of an array */
void print_array(int len, double arr[len])
```

```
{
    printf("\n[ ");
    for (int ielm = 0; ielm < len; ++ielm)
    {
        printf("%f,", arr[ielm]);
    }
    printf(" ]\n");
}

/** print the content of a matrix */
void print_matrix(int row_len, int column_len, double mat[
   row_len][column_len])
{
    printf("\n[ ");
    for (int irow = 0; irow < row_len; ++irow)
    {
        for (int icolumn = 0; icolumn < column_len; ++
           icolumn)
        {
            printf("%f,", mat[irow][icolumn]);
        }
        if (irow != row_len - 1)
        {
            printf("\n");
        }
    }
    printf(" ]\n");
```

```
}

/** Solve the given linear system */
void solve_linear_system(int num, double arr[num], double
    mat[num][num])
{
    // Applying Gauss Jordan Elimination
    for (int i = 0; i < num - 1; i++)
    {
        for (int j = i + 1; j < num; j++)
        {
            float ratio = mat[j][i] / mat[i][i];
            for (int k = 0; k < num; k++)
            {
                if (k == i)
                {
                    mat[j][k] = 0;
                }
                else
                {
                    mat[j][k] = mat[j][k] - (mat[i][k] *
                        ratio);
                }
            }
            arr[j] = arr[j] - (arr[i] * ratio);
        }
    }
```

```
    // Solve the result
    for (int i = num - 1; i >= 0; i--)
    {
        for (int j = num - 1; j >= i; j--)
        {
            if (j == i)
            {
                arr[i] = arr[i] / mat[i][j];
            }
            else
            {
                arr[i] = arr[i] - (mat[i][j] * arr[j]);
            }
        }
    }
}

/** Calculate the lu decomposition of the given matrix */
void ludecomposition(int num, double mat[num][num], double
   lower[num][num], double upper[num][num])
{
    for (int i = 0; i < num; i++)
    {
        for (int j = 0; j < num; j++)
        {
            if (j < i)
            {
```

```
            lower[j][i] = 0;
        }
        else
        {
            lower[j][i] = mat[j][i];
            for (int k = 0; k < i; k++)
            {
                lower[j][i] = lower[j][i] - lower[j][k]
                    * upper[k][i];
            }
        }
    }
    for (int j = 0; j < num; j++)
    {
        if (j < i)
        {
            upper[i][j] = 0;
        }
        else if (j == i)
        {
            upper[i][j] = 1;
        }
        else
        {
            upper[i][j] = mat[i][j] / lower[i][i];
            for (int k = 0; k < i; k++)
            {
```

```
                    upper[i][j] = upper[i][j] - ((lower[i][
                        k] * upper[k][j]) / lower[i][i]);
                }
            }
        }
    }
}

void _inverse(int col, int num, double invmat[num][num],
    double lower[num][num], double upper[num][num])
{
    double x[num];
    double y[num];
    double b[num];

    b[col] = 1;

    for (int i = 0; i < num; i++)
    {
        y[i] = b[i] / lower[i][i];
        for (int k = 0; k < num; k++)
        {
            b[k] -= lower[k][i] * y[i];
        }
    }

    for (int i = num - 1; i >= 0; i--)
```

```
    {
        x[i] = y[i] / upper[i][i];
        for (int k = i - 1; k >= 0; k--)
        {
            y[k] -= upper[k][i] * x[i];
        }
    }

    // filling invmat
    for (int k = 0; k < num; k++)
    {
        invmat[k][col] = x[k];
    }
}

/** Calculate the inverse of the given matrix. */
void inverse(int num, double mat[num][num], double invmat[
    num][num])
{
    double lower[num][num];
    double upper[num][num];
    ludecomposition(num, mat, lower, upper);

    for (int col = 0; col < num; col++)
    {
        _inverse(col, num, invmat, lower, upper);
    }
```

```
}

int main()
{
    // Solving a linear system
    int num_eq = 200;
    int arr_len = num_eq;
    int mat1_rownum = num_eq;
    int mat1_colnum = num_eq;

    double mat1[mat1_rownum][mat1_colnum];
    random_matrix(mat1_rownum, mat1_colnum, mat1);

#ifndef __OPTIMIZE__
    printf("A matrix");
    print_matrix(mat1_rownum, mat1_colnum, mat1);
#endif

    double arr[arr_len];
    random_array(arr_len, arr);

#ifndef __OPTIMIZE__
    printf("b array");
    print_array(arr_len, arr);
#endif

    solve_linear_system(num_eq, arr, mat1);
```

```
#ifndef __OPTIMIZE__
    printf("the anwser (x)");
    print_array(arr_len, arr);
#endif

    // Calculating the inverse of a matrix
    int mat2_num = 100;

    double mat2[mat2_num][mat2_num];
    random_matrix(mat2_num, mat2_num, mat2);

    double invmat[mat2_num][mat2_num];

#ifndef __OPTIMIZE__
    printf("the original matrix");
    print_matrix(mat2_num, mat2_num, mat2);
#endif

    inverse(mat2_num, mat2, invmat);

#ifndef __OPTIMIZE__
    printf("the inverse matrix");
    print_matrix(mat2_num, mat2_num, invmat);
#endif
}
```

References

[1] E. Perl, "Test Requirements for Launch, Upper-Stage, and Space Vehicles," *Aerospace Report No. TR-2004 (8583)-1, Rev. A., The Aerospace Corporation*, vol. 6, 2006.

[2] A. Yahyaabadi, M. Driedger, N. Turenne, S. Connell, V. Parthasarathy, A. Carvey, V. Platero, J. Campos, M. Rososhansky, and P. Ferguson, "ManitobaSat-1: A Novel Approach for Technology Advancement," *IEEE Potentials*, vol. 39, no. 4, pp. 17–23, jul 2020. [Online]. Available: https://ieeexplore.ieee.org/document/9132729/

[3] S. Kaufman and R. Segal, "Preparing for the space innovations of today and tomorrow," Tech. Rep. [Online]. Available: https://bit.ly/3Be6fX3

[4] N. Mohajerin and S. L. Waslander, "Multi-Step Prediction of Dynamic Systems with Recurrent Neural Networks," pp. 1–13, 2018. [Online]. Available: http://arxiv.org/abs/1806.00526

[5] M. Wilde, C. Clark, and M. Romano, "Historical survey of kinematic and dynamic spacecraft simulators for laboratory experimentation of on-orbit proximity maneuvers," *Progress in Aerospace Sciences*,

vol. 110, no. August, p. 100552, 2019. [Online]. Available: https://doi.org/10.1016/j.paerosci.2019.100552

[6] P. A. Ferguson, "Quantifying and Modelling Adaptive Astronaut Movement: Motion Strategies for Long-Duration Spaceflight Missions," Ph.D. dissertation, Massachusetts Institute Of Technology, 2006. [Online]. Available: https://dspace.mit.edu/handle/1721.1/36176

[7] R. Zappulla, J. Virgili-Llop, C. Zagaris, H. Park, and M. Romano, "Dynamic Air-Bearing Hardware-in-the-Loop Testbed to Experimentally Evaluate Autonomous Spacecraft Proximity Maneuvers," *Journal of Spacecraft and Rockets*, vol. 54, no. 4, pp. 825–839, jul 2017. [Online]. Available: https://arc.aiaa.org/doi/10.2514/1.A33769

[8] A. G. Ledebuhr, L. C. Ng, M. S. Jones, B. A. Wilson, R. J. Gaughan, E. F. Breitfeller, W. G. Taylor, J. A. Robinson, D. R. Antelman, and D. P. Nielsen, "Micro-satellite ground test vehicle for proximity and docking operations development," in *2001 IEEE Aerospace Conference Proceedings (Cat. No. 01TH8542)*, vol. 5. IEEE, 2001, pp. 2493–2504.

[9] D. Jung and P. Tsiotras, "A 3-dof experimental test-bed for integrated attitude dynamics and control research," in *AIAA guidance, navigation, and control conference and exhibit*, 2003, p. 5331.

[10] H. G. Hatch Jr, "Rendezvous Docking Simulator," *A Compilation of Recent Research Related to the Apollo Mission, NASA TM X-890*, pp. 187–192, 1963.

[11] E. R. Long, J. E. Pennington, and P. L. Deal, *Remote pilot-controlled docking with television.* National Aeronautics and Space Administration, 1965.

[12] R. H. Tolson, "Lunar Orbit Landing Sites and Stay Times. A Compilation of Recent Research Related to the Apollo Mission," *NASA TM x-890*, pp. 1–9, 1963.

[13] H. G. Hatch, *Dynamic simulation of lunar module docking with Apollo command module in lunar orbit.* National Aeronautics and Space Administration, 1967.

[14] P. R. Lawson, "The terrestrial planet finder," in *2001 IEEE Aerospace Conference Proceedings (Cat. No. 01TH8542)*, vol. 4. IEEE, 2001, pp. 4–2005.

[15] E. M. Kong, A. Saenz-Otero, S. Nolet, D. S. Berkovitz, D. W. Miller, and S. W. Sell, "SPHERES as Formation Flight Algorithm Development and Validation Testbed: Current Progress and Beyond," 2004.

[16] T. Bresciani, "Modelling, Identification and Control of a Quadrotor Helicopter," *Master's Thesis*, no. October, pp. 1–184, 2008.

[17] K. Nonami, F. Kendoul, S. Suzuki, W. Wang, and D. Nakazawa, *Autonomous Flying Robots.* Tokyo: Springer Japan, 2010. [Online]. Available: http://link.springer.com/10.1007/978-4-431-53856-1

[18] M. Elsamanty, A. Khalifa, M. Fanni, A. Ramadan, and A. Abo-Ismail, "Methodology for identifying quadrotor parameters, attitude estima-

tion and control," *2013 IEEE/ASME International Conference on Advanced Intelligent Mechatronics: Mechatronics for Human Wellbeing, AIM 2013*, pp. 1343–1348, 2013.

[19] L. Derafa, T. Madani, A. Benallegue, and I. V. Fin, "Dynamic Modelling and Experirnental Identification of Four Rotors Helicopter Pararneters," pp. 1834–1839, 2006.

[20] I. Stanculeanu and T. Borangiu, "Quadrotor Black-Box System Identification," pp. 350–353, 2011.

[21] N. Abas, A. Legowo, and R. Akmeliawati, "Parameter identification of an autonomous quadrotor," *2011 4th International Conference on Mechatronics: Integrated Engineering for Industrial and Societal Development, ICOM'11 - Conference Proceedings*, no. May, pp. 1–8, 2011.

[22] N. Abas, A. Legowo, Z. Ibrahim, N. Rahim, and A. M. Kassim, "Modeling and System Identification using Extended Kalman Filter for a Quadrotor System," *Applied Mechanics and Materials*, vol. 313-314, pp. 976–981, 2013. [Online]. Available: https://www.scientific.net/AMM.313-314.976

[23] R. W. Beard, "Quadrotor Dynamics and Control Rev 0.1," *Brigham Young University*, no. June, pp. 1–48, 2008.

[24] P. Lyu, S. Bao, J. Lai, S. Liu, and Z. Chen, "A dynamic model parameter identification method for quadrotors using flight data," *Proceedings of the Institution of Mechanical Engineers, Part G: Journal of Aerospace Engineering*, vol. 0, no. 0, pp. 1–13, 2018.

[25] I. Sa and P. Corke, "System identification, estimation and control for a cost effective open-source quadcopter," *Proceedings - IEEE International Conference on Robotics and Automation*, pp. 2202–2209, 2012.

[26] M. Rich, "Model development, system identification, and control of a quadrotor helicopter," 2012. [Online]. Available: https://lib.dr.iastate.edu/etd/12770/

[27] A. Chovancová, T. Fico, E. Chovanec, and P. Hubinský, "Mathematical modelling and parameter identification of quadrotor (a survey)," *Procedia Engineering*, vol. 96, pp. 172–181, 2014.

[28] X. Zhang, X. Li, K. Wang, and Y. Lu, "A survey of modelling and identification of quadrotor robot," *Abstract and Applied Analysis*, vol. 2014, 2014.

[29] M. Bisheban and T. Lee, "Computational geometric identification for quadrotor dynamics in wind fields," *1st Annual IEEE Conference on Control Technology and Applications, CCTA 2017*, vol. 2017-Janua, pp. 1153–1158, 2017.

[30] D. M. Filatov, A. I. Friedrich, and A. V. Devyatkin, "Parameters identification of thrust generation subsystem for small unmanned aerial vehicles," *Proceedings of 2017 IEEE 2nd International Conference on Control in Technical Systems, CTS 2017*, pp. 381–383, 2017.

[31] J. E. Angarita, K. Schroeder, and J. Black, "Quadrotor Model Generation using System Identification Techniques," in *2018 AIAA Modeling and Simulation Technologies Conference*, no. January.

Reston, Virginia: American Institute of Aeronautics and Astronautics, jan 2018, pp. 1–7. [Online]. Available: https://arc.aiaa.org/doi/10.2514/6.2018-1917

[32] S. H. Cho, S. Bhandari, F. C. Sanders, M. Tischler, and K. Cheung, "System Identification and Controller Optimization of Coaxial Quadrotor UAV in Hover," in *AIAA Scitech 2019 Forum*, vol. 62, no. 4. Reston, Virginia: American Institute of Aeronautics and Astronautics, jan 2019, pp. 1–9. [Online]. Available: http://www.ingentaconnect.com/content/10.4050/JAHS.62.042007https://arc.aiaa.org/doi/10.2514/6.2019-1075

[33] C. Dube and J. O. Pedro, "Modelling and Closed-Loop System Identification of a Quadrotor-Based Aerial Manipulator," *Journal of Physics: Conference Series*, vol. 1016, no. 1, 2018.

[34] S. Yang, L. Xi, J. Hao, Y. Zhao, Y. Yang, and W. Wang, "Aerodynamic parameters identification and adaptive LADRC attitude control of quadrotor model," no. November, 2018.

[35] W. Liu, X. Huo, J. Liu, and L. Wang, "Parameter Identification for a Quadrotor Helicopter Using Multivariable Extremum Seeking Algorithm," *International Journal of Control, Automation and Systems*, vol. 16, no. 4, pp. 1951–1961, 2018.

[36] M. Cunningham and J. E. Hubbard, "Open Loop System Identification of a Small Multirotor Vehicle with an Active Feedback Control

System," *2018 Atmospheric Flight Mechanics Conference*, 2018. [Online]. Available: https://arc.aiaa.org/doi/10.2514/6.2018-3475

[37] C. Wang, "Parameter Identification Method for Quad-rotor UAV Based on the Lyapunov-MARI Algorithm," *2018 5th International Conference on Information, Cybernetics, and Computational Social Systems (ICCSS)*, pp. 311–316, 2018.

[38] T. Jiang, J. Li, and K. Huang, "Longitudinal parameter identification of a small unmanned aerial vehicle based on modified particle swarm optimization," pp. 865–873, jun 2015.

[39] G. Brian and E. A. Morelli, "Rapid automated aircraft simulation model updating from flight data," *14th Australian International Aerospace Congress, AIAC 2011*, 2011.

[40] Q. Bian, K. Zhao, X. Wang, and R. Xie, "System Identification Method for Small Unmanned Helicopter Based on Improved Particle Swarm Optimization," *Journal of Bionic Engineering*, vol. 13, no. 3, pp. 504–514, jul 2016.

[41] J. Krizan, L. Ertl, M. Bradac, M. Jasansky, and A. Andreev, "Automatic code generation from Matlab/Simulink for critical applications," *Canadian Conference on Electrical and Computer Engineering*, pp. 1–6, 2014.

[42] S. Gang, C. Dacheng, H. Jingfeng, and H. Jun-wei., "Research on three-axis six-DOF shaking table based on rapid prototyping of DSP algorithms using SIMULINK," *2008 2nd International Symposium on Sys-*

tems and Control in Aerospace and Astronautics, ISSCAA 2008, pp. 1–6, 2008.

[43] "IDNEO Develops Embedded Computer Vision and Machine Learning Algorithms for Interpreting Blood Type Results." [Online]. Available: https://bit.ly/3C7VJC0

[44] "Respiri Develops Mobile App for Wheeze Detection and Asthma Management," 2019. [Online]. Available: https://www.mathworks.com/company/user_stories/respiri-develops-mobile-app-for-wheeze-detection-and-asthma-management.html

[45] "VivaQuant Accelerates Development and Validation of Embedded Device for Ambulatory ECG Sensing," 2019. [Online]. Available: https://www.mathworks.com/tagteam/78014_92143v00_VivaQuant_UserStory_final.pdf

[46] "Delphi Develops Radar Sensor Alignment Algorithm for Automotive Active Safety System," 2019. [Online]. Available: https://www.mathworks.com/tagteam/81084_92225v00_DelphiAutomotive_UserStory_final.pdf

[47] E. Denney and S. Trac, "A software safety certification tool for automatically generated guidance, navigation and control code," *IEEE Aerospace Conference Proceedings*, 2008.

[48] J. Côté and J. de Lafontaine, "Magnetic-Only Orbit and Attitude Estimation Using the Square-Root Unscented Kalman Filter: Appli-

cation to the PROBA-2 Spacecraft," in *AIAA Guidance, Navigation and Control Conference and Exhibit*, no. August. Reston, Virigina: American Institute of Aeronautics and Astronautics, aug 2008, pp. 1–24. [Online]. Available: https://arc.aiaa.org/doi/10.2514/6.2008-6293

[49] D. H. Zarrinkoub, *Understanding LTE with MATLAB®*. Chichester, UK: John Wiley Sons, Ltd, feb 2014. [Online]. Available: http://doi.wiley.com/10.1002/9781118443446

[50] R. M. Murray, Z. Li, and S. S. Sastry, *A Mathematical Introduction to Robotic Manipulation*. CRC Press, dec 2017, vol. 29. [Online]. Available: https://www.taylorfrancis.com/books/9781351469791

[51] D. Mellinger, M. Shomin, and V. Kumar, "Control of Quadrotors for Robust Perching and Landing," in *International Powered Lift Conference*, 2010.

[52] R. Mahony, V. Kumar, and P. Corke, "Multirotor aerial vehicles: Modeling, estimation, and control of quadrotor," *IEEE Robotics and Automation Magazine*, vol. 19, no. 3, pp. 20–32, 2012.

[53] F. Riether, *Agile quadrotor maneuvering using tensor-decomposition-based globally optimal control and onboard visual-inertial estimation*, 2016, no. 2013. [Online]. Available: http://hdl.handle.net/1721.1/106777

[54] D. Mellinger, Q. Lindsey, M. Shomin, and V. Kumar, "Design, modeling, estimation and control for aerial grasping and manipulation," *IEEE*

International Conference on Intelligent Robots and Systems, pp. 2668–2673, 2011.

[55] D. Mellinger, A. Kushleyev, and V. Kumar, "Mixed-integer quadratic program trajectory generation for heterogeneous quadrotor teams," *Proceedings - IEEE International Conference on Robotics and Automation*, pp. 477–483, 2012.

[56] S. Ruder, "An overview of gradient descent optimization algorithms," *arXiv preprint arXiv:1609.04747*, 2016.

[57] R. Schoenberg, "Optimization with the quasi-newton method," *Aptech Systems, Inc. Maple Valley, WA*, 2001.

[58] J. Kennedy and R. Eberhart, "Particle swarm optimization," in *Proceedings of ICNN'95 - International Conference on Neural Networks*, vol. 4, 1995, pp. 1942–1948 vol.4.

[59] Digilent, "Zybo Z7 Board Reference Manual," pp. 1–30, 2017. [Online]. Available: https://reference.digilentinc.com/reference/programmable-logic/zybo-z7/reference-manual

[60] Xilinx, "Zynq-7000 SoC Data Sheet: Overview," pp. 1–25, 2019. [Online]. Available: https://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf