INSTITUTO TECNOLÓGICO
Y DE ESTUDIOS SUPERIORES DE OCCIDENTE

Reconocimiento de validez oficial de estudios de nivel superior según acuerdo secretarial 15018, publicado en el Diario Oficial de la Federación el 29 de noviembre de 1976.

Departamento de Electrónica, Sistemas e Informática

DOCTORADO EN CIENCIAS DE LA INGENIERÍA

SISTEMA DE CONTROL EMBEBIDO PARA MULTI-ROTORES CONSIDERANDO DINÁMICA DE MOTORES

Tesis que para obtener el grado de

DOCTOR EN CIENCIAS DE LA INGENIERÍA

presenta: Walter Alejandro Mayorga Macías

Director de tesis: Dr. Luis Enrique González Jiménez

Co-director de tesis: Dr. Luis Fernando Luque Vega

Tlaquepaque, Jalisco. Diciembre de 2020

DOCTOR EN CIENCIAS DE LA INGENIERÍA (2020) ITESO, Tlaquepaque, Jal., México

TÍTULO: **Sistema de Control Embebido para Multi-Rotores Considerando Dinámica de Motores**

AUTOR: Walter Alejandro Mayorga Macías
Ingeniero en Mecatrónica (ITESM, México)
Maestro en Ciencias en Ingeniería Electrónica y Computación (Universidad de Guadalajara, México)

DIRECTOR DE TESIS: Luis Enrique González Jiménez
Departamento de Electrónica, Sistemas e Informática, ITESO
Ingeniero en Electrónica (ITSON, México)
Maestro en Ingeniería Eléctrica (CINVESTAV Guadalajara, México)
Doctor en Ingeniería Eléctrica (CINVESTAV Guadalajara, México)

NÚMERO DE PÁGINAS: xxvii, 184

ITESO – The Jesuit University of Guadalajara

Department of Electronics, Systems, and Informatics

DOCTORAL PROGRAM IN ENGINEERING SCIENCES

EMBEDDED CONTROL SYSTEM FOR A MULTI-ROTOR CONSIDERING MOTOR DYNAMICS

Thesis to obtain the degree of

DOCTOR IN ENGINEERING SCIENCES

Presents: Walter Alejandro Mayorga-Macías

Thesis Director: Dr. Luis Enrique González-Jiménez

Thesis Co-director: Dr. Luis Fernando Luque-Vega

Tlaquepaque, Jalisco, Mexico
December 2020

PHD IN ENGINEERING SCIENCES (2020) ITESO, Tlaquepaque, Jal., Mexico

TITLE: **Embedded Control System for Multi-Rotors Considering Motor Dynamics**

AUTHOR: Walter Alejandro Mayorga-Macías
Bachelor's degree in mechatronics engineering (ITESM, Mexico)
Master's degree in electronics and computer engineering (University of Guadalajara, Mexico)

THESIS DIRECTOR: Luis Enrique González-Jiménez
Department of Electronics, Systems, and Informatics, ITESO
Bachelor's degree in electronics engineering (ITSON, Mexico)
Master's degree in electrical engineering (CINVESTAV Campus Guadalajara, Mexico)
Ph.D. degree in electrical engineering (CINVESTAV Campus Guadalajara, Mexico)

NUMBER OF PAGES: xxvii, 184

To my family, for their love, support and
guidance in this journey

Resumen

Este trabajo tiene como finalidad el diseño y la implementación embebida de un esquema de control para el seguimiento de trayectorias para multi-rotores. Como punto inicial se aborda el estado del arte en cuanto a desarrollo de aeronaves de despegue y aterrizaje vertical (VANT), así como los tipos de VANT actuales. Con ello definido, el documento describe la problemática sobre los esquemas de hardware y software para el desarrollo e investigación en este tipo de dispositivos. Así, se propone un esquema de control para estas aeronaves, el cual incluye un doble lazo de control que corresponden al control de pose de la aeronave y el control de velocidad de los actuadores.

La propuesta de investigación contempla el modelado matemático del sistema usando aproximaciones de ángulos pequeños, lo cual simplifica las ecuaciones de estado al reducir el impacto de la matriz de rotación sobre el estado del vehículo. Esta simplificación representa una ventaja durante el posterior proceso de diseño de los controladores. Este modelo es validado por medio de simulaciones en lazo abierto (sin considerar el controlador) mediante la herramienta de simulación Matlab/Simulink. Dicho proceso consiste en definir diversas entradas de velocidad en los motores, graficar el comportamiento de los estados del modelo y corroborar que este es el esperado en un vehículo real con las mismas entradas definidas. Dado que el modelo a simular requiere de los parámetros del sistema, los métodos de obtención de estos son desarrollados ya que, comúnmente, dicha información no es proporcionada por los fabricantes de los motores sin escobillas o los desarrolladores de las estructuras comerciales para VANTs.

Una vez validado el modelo dinámico del VANT a controlar, los lazos externo e interno de control por retroalimentación de la variable de error son diseñados mediante los esquemas *backstepping* y PID (Proporcional-Integral-Derivativo). Además, la demostración de estabilidad para cada esquema de control en lazo cerrado es realizada de dos maneras: de forma algebraica usando el teorema de Lyapunov, y de forma numérica mediante resultados en simulación.

Para la implementación en tiempo real uno de los esquemas de control diseñados es desarrollado en un sistema embebido y se realizan experimentos utilizando un VANT comercial con la misma configuración contemplada durante el modelado matemático. Además, se describen las arquitecturas de hardware y software usadas en el vehículo real para la implementación de los

algoritmos de control y acondicionamiento de las señales obtenidas por los sensores. Los resultados de dichos experimentos son reportados y analizados, y las conclusiones obtenidas son presentadas.

Durante el proceso de validación es pertinente el uso de dispositivos de prueba, que de manera segura permitan al desarrollador la sintonización de ganancias del sistema para el control eficiente de la aeronave. Esto abre la pauta a la consolidación de un sistema completo de validación el cual permita la modificación y evaluación de parámetros y nuevas propuestas de investigación, ya sea para proporcionar habilidades de inteligencia a dispositivos VANT, características de detección de fallos, agregar soporte a nuevas tecnologías, o para el desarrollo de metodologías educativas.

Summary

This dissertation aims to design and implement an embedded control scheme for trajectory tracking in multi-rotors. The first objective to tackle is the state of the art about the vertical take-off and landing (VTOL) devices, furthermore, the document defines the available VTOL types in the field. With the previously mentioned, the thesis defines the problematic about the available hardware and software schemes for research and development for unmanned aerial vehicles (UAVs). Therefore, a control scheme proposal is described for this type of aircrafts, which includes a nested double loop cycle that considers the control of the vehicle and the speed control of the actuators.

The research proposal considers the hypothesis of small angles approximation, which simplifies the mathematical analysis of the system, and the obtaining of a state-space model, by reducing the impact of the rotation matrix over the vehicle. The simplification of the model represents an advantage during the process of controller design for the system. This model is validated with the use of open-loop simulations (without considering the controller) in Matlab/Simulink. Such process consists in defining different motor speeds inputs, displaying the behavior of the model states and corroborate that the model behavior is the expected when compared with the real system. Given that the model requires the parameters of the system during the simulations, the methods for obtaining such information are developed since that data is not commonly provided by the brushless motors manufacturers or VTOL structures developers.

Once the dynamic model of the VTOL to control is validated, the external and internal control loops with error feedback are designed using the backstepping and PID (Proportional-Integral-Derivative) schemes. Furthermore, the stability validation for each scheme is developed in two ways: by mathematical demonstration using Lyapunov's theorem, and by numeric results from simulation.

For the real-time implementation, one of the designed control schemes is implemented in an embedded system, and numerous experiments are performed using a commercial VTOL with the same configuration and motor arrangement considered during the mathematical modeling process. Furthermore, the hardware and software architectures used are described for the implementation of the control algorithms and for the sensor signal conditioning. The results for the experiments are reported and analyzed, and the respective conclusions are presented.

During the validation process, the use of test devices is pertinent with the purpose of allowing the developer to tune the system controller gains safely. This opens the door to the consolidation of a complete validation system that allows the modification and evaluation of new parameters and research proposals, with the purpose of adding intelligence abilities to VTOL, providing fault detection capabilities, adding support for new technologies, or for developing educational methodologies.

Acknowledgements

The author wishes to express his sincere appreciation to Dr. Luis Enrique González-Jiménez, professor of the Department of Electronics, Systems, and Informatics at ITESO, for his encouragement, expert guidance and keen supervision as doctoral thesis director throughout the course of this work. The author offers his gratitude to Dr. Luis Fernando Luque-Vega from UVM, for his support as doctoral thesis co-director during the development of this work. He also thanks Dr. Marco Antonio Meza-Aguilar, Dr. Juan José Ley-Rosas, and Dr. Riemann Ruiz-Cruz, members of his Ph.D. Thesis Committee, for their interest, assessment, and suggestions.

The author has greatly benefited from working with MATLAB developed by The MathWorks Inc, with ArduPilot developed by the ArduPilot Dev Team, and with Pixhawk developed by 3D Robotics.

It is the author's pleasure to acknowledge the fruitful collaboration and stimulating discussions with his colleagues of the Department of Electronics, Systems, and Informatics at ITESO: Dr. Marco Antonio Meza-Aguilar, Guilhem Jouhet, and Marco Antonio López-Vidaurri.

The author gratefully acknowledges the financial assistance through a scholarship granted by the *Consejo Nacional de Ciencia y Tecnología* (CONACYT), Mexican Government, as well as the financial support provided by Intel Corporation.

Finally, special thanks are due to my family: my mother Irma Mayorga, and my father Adrián Fernández, for their understanding, patience, and continuous loving support.

Contenido

Contents

List of Figures

List of Tables

Introduction

The drone terminology is commonly used nowadays, and the term involves numerous areas and technologies in their implementation. There are multiple classes of unmanned vehicles in existence, from hand-launched sub-scale models to full-size long types. Most of them are remotely piloted, but there are some others that are operated in certain controlled environments going in a gradual progression towards full autonomy. Just as a primary approach, drones in the aerial realm are commonly named unmanned aircraft systems (UAS), unmanned aerial vehicles or uninhabited air vehicle (UAV), Vertical Take-Off and Landing (VTOL) devices, multi-rotors, or multi-copters, just to name a few.

In the simplest way of view, a UAV is an aircraft with no aircrew, therefore, it is replaced by a control computer system and radio-link for remote controllability or inspection. In a more realistic manner, it is a complex system that is properly designed and is comprised of a control station which serves as a control interface between the operator and the rest of the technology involved. In a listed form, the UAV implies the aircraft itself, payloads, support sub-systems, communication sub-systems, transport sub-systems, and others.

There are differences between a drone aircraft and a UAV. While they are normally named the same, they are not equal. Drone aircrafts are basically not intelligent devices, which are normally launched with a pre-programmed mission, on a pre-programmed course, and a return to base; communication is not a feature, and the results of the mission are not obtained until the vehicle is recovered at the base. On the other hand, the UAVs have a certain grade of automatic intelligence, being capable of communicating with its controller, and normally data return is provided along with primary state information such as position, orientation, and altitude.

Origins of UAV Research and Development

The idea of a vertical flight aircraft can be traced back in time about 400 century B.C. with the Chinese tops, which were basically feathers attached at the end of a stick that was rapidly spun between the hands to generate lift and then released for free flight. After that, the first solid reference with respect to multi-rotors places at the time of the multidisciplinary Leonardo da Vinci,

in 1483. The aerial-screw invention that he wrote in the Codex Atlanticus with the following phrase: “I have discovered that a screw-shaped device such as this, if it is well-made from starched linen, will rise in the air if turned quickly”, described what has been named the helicopter’s ancestor [Leishman-06], [AIAA-18].

Several centuries passed until new developments were reached, and it was not until 1783 that Launoy and his mechanic, Bienvenu, presented a coaxial model of a simple helicopter powered by the tension in a crossbow [Lemos-07]. That invention started the stir within the scientific circles and in 1786 the French mathematician A. J. P. Paucton published in 1786 the “*Theorie de la vis D’Archimede*”, which is the first scientific paper that talks about the problem of rotating wings.

Continuing with the development of the VTOL devices, in 1804, Sir George Cayley constructed a whirling-arm device, which is one of the first scientific attempts to study the aerodynamic forces produced by lifting wings. Later, in the period between 1809 and 1810, he published a three-part paper that placed the foundations of modern aerodynamics.

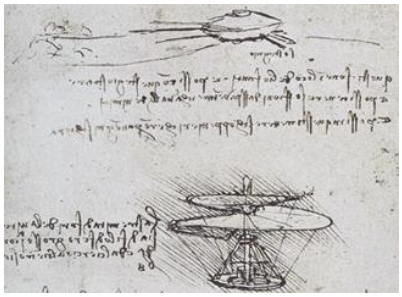
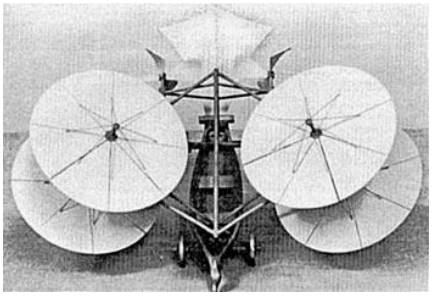
a)  b) 

Fig. I a) Da Vinci’s aerial screw, b) Cayley’s Aerial Carriage.

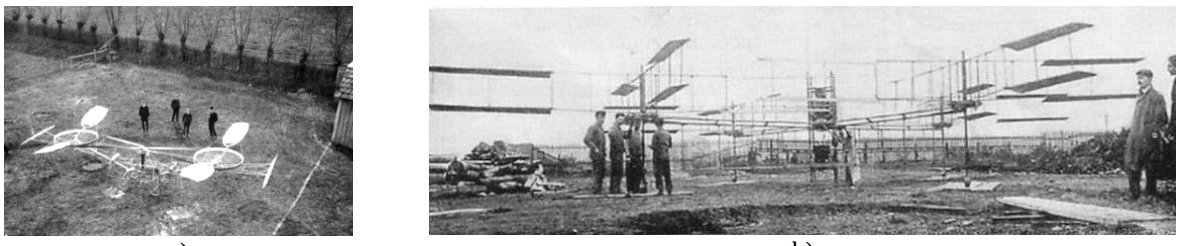
On August 22, 1849, Australia launched around 200 balloons against Venice, Italy, and the balloons were armed with explosives controlled by timed fuses. This records the first UAV in history. The lack of control of the devices caused minimal damage to the city, instead, many of the unpiloted balloons blew back towards Australians [RPAV-03].

During the time of the early 1860s, Ponton d’Amecourt from France, flew a several of small steam-powered helicopter models. Other noticeable VTOL models from that time are the design of Bright in 1861, and the twin-rotor steam-driven model of E. Dieuaide in 1877. Within that period, in 1874, Wilhelm von Achenbach introduced the idea of a sideward thrusting tail rotor that counteracts the torque reaction from the main rotor. In 1869, A. Lodygin developed the Russian “electroplane” concept, using a rotor for lift and a propeller for propulsion and control. Later, Enrico Forlanini from Italy, build another type of flying steam-driven helicopter model in

1878.

In 1880, Thomas Alva Edison experimented several small helicopter models in the United States, having the most promising results with an electric motor design. His scientific approaches proved that both high-efficiency rotors and high-power engines were required for successful flights to be achieved. In 1910, he patented a rather cumbersome looking full-scale helicopter concept with box-kite-like blades.

At the year of 1907, the French Paul Cornu constructed a vertical flight machine that carried a human off the ground for the first time. The engines that supplied the power to the rotors were 24-hp gasoline motors and a gear transmission, where each rotor had two blades, and each actuator rotated in opposite directions to cancel torque reaction. In the same period, the Breguet brothers conducted helicopter experiments by the guidance of Professor Charles Richet, with a quad-rotor called Gyroplane No. 1. It carried a pilot off the ground, but it was underpowered with a 40-hp motor, never flew completely free and lacked stability and proper control.

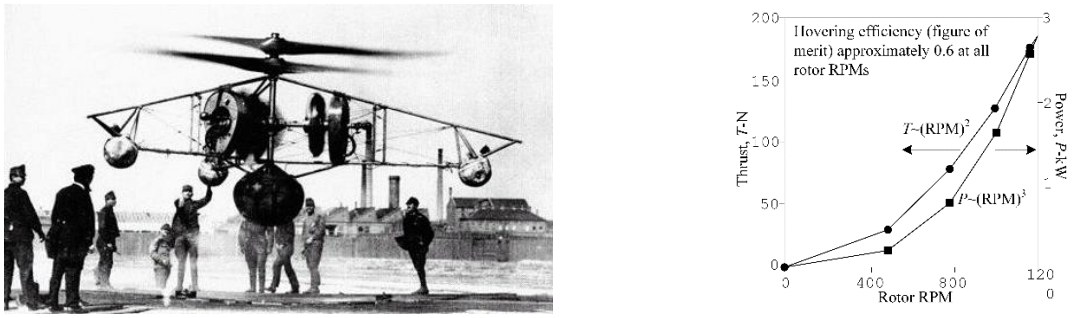


a) b)

Fig. II a) Cornu's aircraft, b) Breguet-Richet Gyroplane No. 1.

In the early 1900s, Igor Ivanovich Sikorsky and Boris Yuryev did some unsuccessful experiments due to vibration problems and lacked powerful engines but established the concept of the cyclic pitch for rotor control. During that period, several theoretical contributions on the subject about aerodynamics were done by professor Zhukovskii, while at the same time, Stefan Drzewiecki developed a hybrid momentum/blade element concept and published a book entitled "*Des Helices Aerinnes Theorie Generale des Propulseurs*".

By the time of 1917-1920, Stephan Petroczy, with the help of Theodore von Kármán and Wilhelm Zurovec, built a coaxial helicopter PKZ 2 which introduced the term of redundancy for engine failure events. Another significant achievement is that von Kármán provided laboratory test results for the rotors, which used oversized propellers. The results agreed on the elementary rotor theory in which the thrust and power should increase with the square and cube of motor RPM; they also provided the efficiency results for the test, which provided a 60%.



a) b)

Fig. III a) Petroczy-Kármán-Zurovec PKZE helicopter, b) Theodore Von Kármán PKZ 2 motor performance measurements.

By the early 1900s, Emile and Henry Berliner designed and built diverse aircraft but it was not until early 1920s that they had some successful tests with a device that had side-by-side rotors. The differential longitudinal tilt of the rotor shafts provided yaw control. Their rudimentary piloted helicopters attempts are credited as the first helicopter developments in the United States.

At the time that World War I was running, in 1914, the British military used aerial photography in the Battle of Neuve Chapelle to capture around 1,500 sky view maps of the German fortifications in the region. In 1917, the United States Navy developed the aerial torpedo, which has a guidance method consisting of a primitive calculation. Once the wind speed, wind direction, and target distance were determined, the number of revolutions that the engine required to take the missile to target were calculated. In airborne, the device was controlled by a small gyroscope, and the altitude measured by a barometer [Austin-10]. On their side, the United States Army started developing UAV technology in 1916, having successful results by 1918 with the "Kettering Bug" which was a self-flying aerial torpedo created by Orville Wright and Charles F. Kettering [Stamp-13].

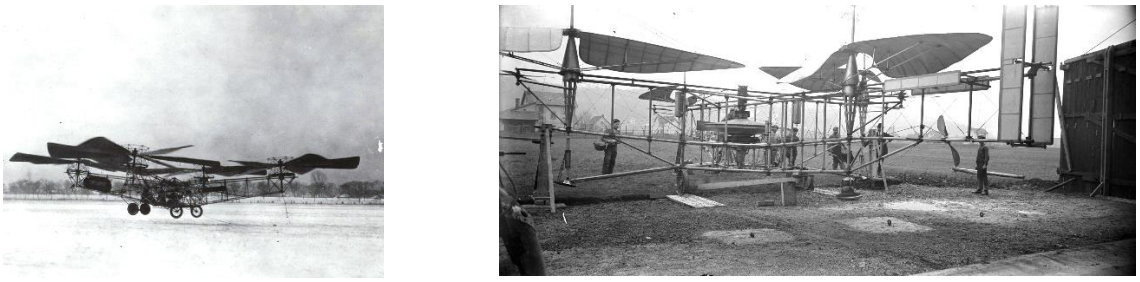
The British Army began its UAV development in 1914 with the aerial target, which that was a radio-controlled pilotless monoplane. Within the same territory, Louis Brennan worked on a helicopter concept that solved the torque reaction problem by powering the rotor with propellers mounted on the blades, and control was achieved using ailerons inboard of the propellers. The lack of stability and control caused the machine to crash on its seventh flight attempt. Later, in 1927, the Royal Navy developed a monoplane capable of carrying a several kg. of load. It was fitted with a radio control for the launch mode, after which the autopilot controlled the flight to a pre-set course.

The Argentinean Raul Pescara built and attempted to fly coaxial helicopters with biplane-

type rotors in the 1920s. His Pescara No. 3 had five sets of biplane blades that were mounted rigidly to the rotor shaft. He was focused on a completely controlled machine, which was achieved through cyclic-pitch changes that were obtained by warping the blades periodically as they rotated, achieving the first successful application of the cyclic-pitch theory. The yaw angle was controlled by the differential collective pitch between the two rotors.

Between 1924 and 1930, A. G. von Baumhauer designed and built a single-rotor helicopter with a tail rotor for controlling torque reaction. It was very difficult to control due to the disconnection between the main and tail rotors. In Austria, the engineer Raoul Hafner designed a single-seat helicopter called R-2 Revoplane, and for control, the machine used swashplate for blade pitch, which became the standard for providing pitch control in modern helicopters.

In 1922, a Russian immigrant in the United States, Georges de Bothezat, built one of the largest helicopters in history. This machine was a quad-rotor with each of the rotors placed at the ends of a cross structure of intersected beams. Regarding control, the device used collective, differential collective and cyclic pitch variations, and the blade pitch design was inspired by Boris Yuryev's configuration. At the same time, Etienne Oehmichen from France built a quad-rotor machine called Oehmichen No. 2, like Bothezat's, but with a few additional rotors for control and propulsion. In 1924, he made reasonable flights and his device proved perfectly maneuverability and stability [Leishman-06].



a) b)

Fig. IV a) Bothezat's aircraft, b) Oehmichen No. 2.

Once aerial vehicles were being studied and developed in a more serious manner, UAVs started gaining more focus and during World War II the development was initiated once again. The next attempts started in 1930, when the United States Navy began experimenting with radio-controlled aircrafts, resulting in the Curtiss N2C-2 drone by 1937. Reginald Denny, during World War II in 1941, created the first remote-controlled aircraft, the Radioplane OQ-2. This was the first mass-produced UAV, which was a breakthrough in manufacturing and supply for drones in

the militia [O'Donnell-17].

It was not until the 1970s that some poor developments were done, and others abandoned due to the time limitations for constructions. Some of them were Lockheed Aquila, the MBLE Epervier, and the Boeing Gull HALE UAS. Another vehicle developed in that period was the Westland Wisp, which was a vertical take-off UAV (VTUAV) designed by the United Kingdom. This was one of the first devices adopting the multi-rotor configuration that solved the problems of launch and recovery.

In 1973, Israel developed the Mastiff UAV and the IAA Scout, and later, in 1982 the Israeli Air Force used this technology over the Syrian Air Force. Later, the United States created the Pioneer UAV Program to fill the gap in the need for inexpensive and unmanned aircraft for operations. In 1986, the United States in joint with Israel created the RQ2 Pioneer, which was a mid-size reconnaissance aircraft. The previous designs were streamlined form factor vehicles; nevertheless, VTOL devices were also developed in the 1980s, being the Canadair CL-227 VTUAV Sentinel one of them. However, after several phases, the aircraft suffered numerous modifications, adopting a streamlined form factor again. Later, the design was reverted to VTOL with the CL-327.

In more recent progress, in 1990s, miniature and micro UAVs were introduced, and GPS and satellite communications freed the UAV limitations of operating within radio tracking range or relying on inaccurate onboard navigation systems. Digital flight control systems were designed and enabled the UAVs to operate within greater ranges using positional accuracy.

In the upcoming years, drones are proposed to increase their capabilities by improving their efficiency and addressing numerous issues, in order to increase their use for civilian tasks such as medicine and package delivery, inspection and maintenance for energy plant facilities, and even first aid lifesavers during emergencies [Strickland-18], [CFA-18].

Types of Multi-Rotors

The types of developed multi-rotor UAVs are numerous. The classification used in this work is based on its rotor configuration, populsion abilities and added capabilities, having two different groups that are defined in the following paragraphs.

Vertical Take-Off and Landing (VTOL) Configuration

This class is also known as rotary-wing UAV or rotorcraft UAV and its advantages are hovering capability and high maneuverability, features that are useful in many robotic missions, especially in civilian applications. Under this category, different configurations are found like the tail rotor, co-axial rotor, tandem rotors, or multi-rotors [Austin-10].

Single Rotor

In this configuration, the weight is counteracted by a small, side-thrusting, tail rotor that typically adds about a further 10% onto the main rotor power demands. One of the disadvantages of this aircraft is that it is extremely asymmetric in all planes, and that adds more complication and complexity to the flight control algorithms. Also important is to mention that the tail rotor is relatively fragile and vulnerable to striking ground objects. Nevertheless, this is configuration is the most popular by the VTOL UAV manufacturers.

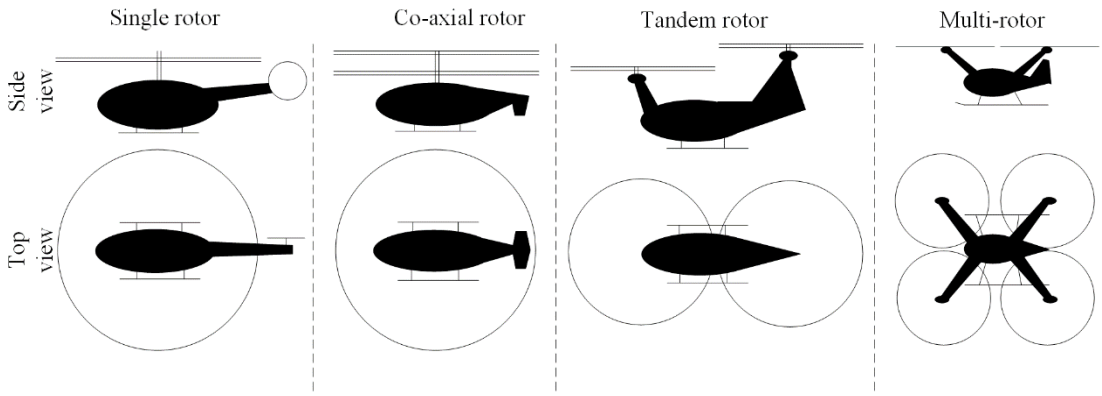


Fig. V Types of VTOL aircrafts

Tandem Rotor

Two small rotors are actuating in this configuration, and even if this sub-type is not the most appropriate for UAVs, the configuration is more symmetric in control than the single rotor and more power efficient. Its small payload volume does not require a long fuselage so that the rotors must be mounted on extended pylons, which is not structurally efficient.

Co-Axial Rotor

This configuration is not generally popular due to its greater height, compared to the other VTOL configurations. Nevertheless, its advantages include almost perfect aerodynamic symmetry, compactness with not vulnerable tail-rotor, power efficiency and versatility for providing alternative body designs for different uses, but each using the same power unit, transmission, and control sub-systems. Therefore, the control system is not more complex than the one in the typical HTOL aircraft. In addition, its response to air turbulence is the lowest from all the VTOL configurations, being zero in most of the cases. However, design limitations may modify this advantage.

Compared to the single rotor, testing measurements proved that under the same environment and similar situations, the co-axial rotors use less power in hover flight due to the less power wasted through swirl energy being left in the rotor downwash. Moreover, the shaft between the two rotors contributes to add extra drag in forwarding flight, and this is regarding the elimination of the tail rotor drag.

Multi-Rotor

Mostly implemented in mini or micro sizes, the goal of this configuration is to remove the complication of rotor-head control systems, applying both cyclic and collective pitch changes to achieve aircraft control. The main control idea is to have rotor blades all fixed in pitch and to achieve thrust changes on each rotor by changing its speed of rotation. The latter is possible by having an individually driven electric motor mounted at the rotor head. Thereby, for moving forward, the rotational speed of the rear rotors would be increased to rotate the aircraft nose down and translate the resulting thrust vector forward. During this operation, the total thrust must be increased to prevent the loss of height, and once in forwarding flight, the rotor speeds must be harmonized again.

Regarding the control algorithm, because of the challenging aerodynamic interference between rotors, it is extremely complicated. And the latter not being enough, the configuration is naturally more gust-sensitive than the other VTOL aircraft and its control response is expected to be slower, meaning that it is difficult to achieve control under laboratory conditions, and more

problematic under urban operations.

Hybrid Configuration

Helicopters have been shown to be the most efficient hover flight aircraft but they are speed limited. For achieving a long-range mission, it is necessary to have the aircraft cruise at a higher speed for having higher speeds and acceptable response time to target. For that reason, the combining of the ability of VTOL and HTOL in a single device is required. Regarding that, various attempts of devices have been made for that necessity.

Convertible Rotor Aircraft

In this category, the rotors are mounted onto each tip of the main wing of an HTOL, and they are positioned in horizontal for vertical flight and tilted 90 degrees for becoming propellers for cruise flight. The wing fixed horizontally is still placed in the fuselage. One alternative of this is to have the wing, the motor and the rotors to be constructed as an assembly, and that complete arrangement to be tilted on the upper fuselage. That is well known as tilt-wing.

Tilting the engines in either tilt-rotor or tilt-wing configuration requires the engines to be operable over a range of 90 degrees, at least, leading with some complications for the fuel and oil delivery to the system, but that complication is more acceptable than replacing the engine to a fixed location and transfer the drive from there to the tilted-rotor system.

Both configurations have been developed successfully, however, the tilt-rotor configuration is more efficient during hover while the others during cruise flight but having a reduced payload capability.

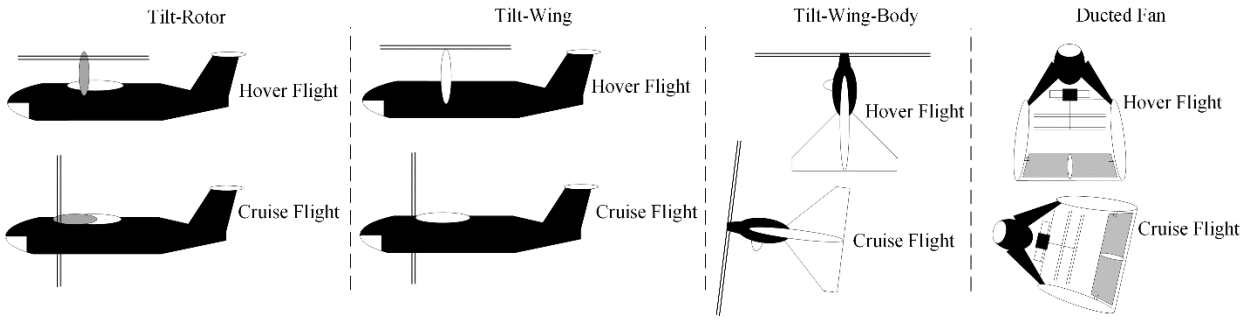


Fig. VI Types of hybrid aircraft

Tilt-Wing-Body Aircraft

The aircraft under this classification is all rotor convertible, which implies critical aerodynamic design for the transition from hover flight to cruise flight. The problem there lies in the ability to maintain the airflow over the wing and achieving attitude control, especially for longitudinal pitch angle. For addressing that problem, helicopter rotors are commonly used instead of propellers for having the cyclic pitch that provides pitch and yaw control. The latter problem becomes more difficult when transitioning from cruise flight to hover and landing.

Ducted Fan Aircraft

This aircraft encloses the thrust mechanism within the duct. The fans used for propulsion have a constrained diameter and high solidity, composed of two contra-rotating elements for minimizing the body rotation caused by the torque produced. Collective and cyclic pitch control are required on the blades to get thrust resulted by changes in fan rotational speed, and for solving the body tilting problem, tilt vanes are used.

The main problem of this kind of vehicles is the attitude control as the vanes may lack of enough force or response to ensure a controllable system.

Jet-Life Aircraft

Vehicles under this configuration are suspended in hover flight on one or more high-speed jets. Other smaller jets are needed for roll and pitch attitude control. For transitioning from hover to forward flight, the jets are rotated backward to provide the element of forward thrust but retaining a vertical component until the fixed-wing develops sufficient lift to sustain the aircraft, at which the jets are effectively horizontal to provide propulsive force only.

Since the system is very expensive in the engine and fuel consumption inefficient, this aircraft is only suitable for high-speed missions and when vertical take-off and landing is required [Nonami-10].

Problem Definition

Multi-rotor UAV control algorithms are a widely researched topic, nevertheless the contributions are commonly provided as mathematical verifications with simulation test cases. The need of functional hardware implementation with an embedded software system is crucial for verification of control schemes. The previous can provide a hardware test-bench for algorithms validation. Furthermore, commercial devices not always provide the electrical and mechanical characteristics required to model the system, which creates a conflict when simulating and manipulating the model since variable unknowns are not desired in the model. For improving the execution, it has been detected that no motor speed feedback is normally received from the control schemes found in bibliography. That creates the problem of not knowing the operation status from the actuators and may cause situations that the flight control cannot solve. Aiming to address this problem, some possible solutions have been found in the bibliography, as the actuator rotation frequency reading by optical-reflective sensors [González-Hernández-12], [Niekerk-15], [Adamo-17], the rotor forces measurement for calculating thrust and local aerodynamics in order to inferred estimate velocity [Davis-17]. The actuator rotator frequency measurement requires high-speed sensing of the sensor output, and the subsequent conversion to frequency, for that, at least, analog-to-digital converters are needed, and a time measurement routine is required for knowing the time-lapse between captures of the toggling. On the other hand, velocity estimation by force sensing requires expensive sensing devices that need to be attached to each rotor, and they provide low sensitive tracking due to force cancelation by the rotor counterpart, requiring fusing along with the inertial measurement unit (IMU) in the aircraft.

As listed previously, the solutions found are reliable, but they require high computation, intrinsic dynamics and numerous and expensive devices attached to each rotor in the multi-rotor configuration.

Solution Proposal

The present dissertation proposes a methodology for obtaining the dynamical parameters of a multi-rotor. These are required for the control design procedure and to simulate the closed-loop system to validate the controller's performance. The overall system provides a baseline for

future control algorithms implementations or new research approaches. In addition to that, the hardware and software are integrated aiming to execute the control algorithms in a real platform for verification of the controller. Moreover, this research proposes the addition of a motor control loop between the output of the flight controller and the brushless actuators. That methodology gives the ability to know the status of the motor by knowing the actual motor speed produced. To effectively design the actuator control loop, a model for the actuator must be obtained. This is proposed by means of transfer functions by defining the applied voltage as the input and the angular speed velocity as the output of the motor.

In order to achieve the previous purposes, the proposal considers real-time current and speed sensors, which are obtained by Hall-effect sensors that provide an analog output for the system to be converted into a current value and by optical reflective sensors [González-Hernández-12] timely tracked for calculating the angular velocity of the actuators. That information feeds the system in a double cycle control loop that maneuvers the vehicle.

This doctoral dissertation is organized as follows:

Chapter 1 introduces the theoretical foundations of the thesis. This includes the dynamical modelling of the vehicle and its actuators, the control schemes used in previous works, the embedded implementations of controllers for multi-rotors and their flight modes. Chapter 2 presents the obtention of the dynamical model of the multi-rotor investigated in this work. Also, it describes the procedure utilized to characterize the parameters of the vehicle and its actuators. Moreover, the chapter presents the open loop simulations of the resulting model as a validation of its correctness. Chapter 3 describes the control design procedure for the vehicle and for the actuators, and the closed-loop system simulations are depicted in Chapter 4. It, also, includes the correspondent analysis and discussion on the obtained results. The methodology of the implementation of the proposed controller on an embedded system is presented in Chapter 5. In addition, the real time experiments are described, and their results are presented and analyzed. The conclusions of the overall developed work are outlined in Chapter 6 and the related future work is depicted. Finally, this thesis includes 6 appendices. Appendix A shows the internal research reports generated during the doctoral studies, and Appendix B lists the publications related with the research developed during the same period. Appendix C describes the code used to simulate the vehicle and the designed controllers in Matlab. The rest of the appendices describes the code generated for the embedded implementation of the proposed controllers for the multi-rotor.

1. Theoretical Framework on Multi-Rotors

Nowadays, multi-rotor UAVs are considered one of the best performing aerial platforms to be used for research, development, and daily use applications. This popularity is due because of their hovering capacity, stabilization capabilities, as well as their reduced mechanical complexity and usually high payload. Most of the late researches focus their efforts in achieving autonomous flight, however, flying autonomy is not a straightforward task if the controller of the vehicle is not considered. While there are numerous control schemes that do not require the knowledge of the vehicle dynamics for maneuvering the aircraft, such as PID, neural networks or fuzzy logic, the lack of known parameters makes them more sensitive to faults and errors in the system, or even susceptible to disturbances.

In this chapter the coverage of the multi-rotor body and motor mathematical model analysis is addressed, with an initial approach to a four-rotor configuration, that can be later easily scaled to another multi-rotor geometry by using the thrust and drag forces combinations matrix M. Moreover, an initial approach analysis of the multiple existing control algorithms is developed to exemplify the current state of the art.

1.1. Mathematical Model of the Vehicle

The body frame of the multi-rotor UAV is normally a symmetric shape structure that can be translated from a point *E* to a point *B*, which are usually defined as the origins of the earth fixed frame E^E and the body fixed frame E^B, respectively.

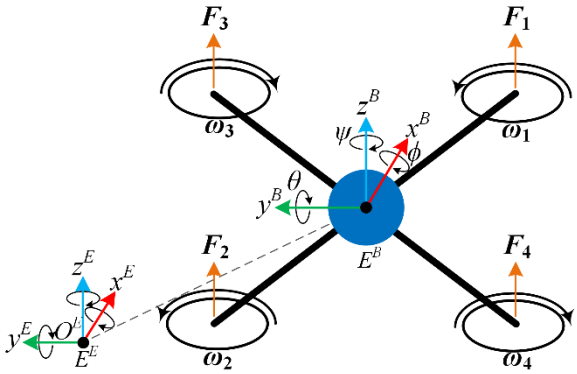


Fig. 1.1 Relative attitude and orientation of the UAV with respect to the fixed frame.

In addition, the attitude of the vehicle is equivalent to the relative orientation of frame E^B with respect to the inertial frame E^E. These geometrical relationships are depicted in Fig. 1.1.

With that consideration, the spatial orientation of the body-fixed frame E^B can be expressed in the earth-fixed reference frame E^E by a rotation matrix R and P from $\{B\}$ to $\{E\}$ with the following transformation and rotation velocity matrices [Luque-Vega-14]

$$R_B^E = \begin{bmatrix} \cos\theta\cos\psi & \sin\phi\sin\theta\cos\psi - \cos\phi\sin\psi & \sin\phi\sin\psi + \cos\phi\sin\theta\cos\psi \\ \cos\theta\sin\psi & \cos\phi\cos\psi + \sin\phi\sin\theta\sin\psi & \cos\phi\sin\theta\sin\psi - \sin\phi\cos\psi \\ -\sin\theta & \sin\phi\cos\theta & \cos\phi\cos\theta \end{bmatrix} \tag{1-1}$$

$$P_B^E = \begin{bmatrix} 1 & 0 & -\sin\theta \\ 0 & \cos\phi & \sin\phi\cos\theta \\ 0 & -\sin\phi & \cos\phi\cos\theta \end{bmatrix} \tag{1-2}$$

where ϕ, θ and ψ are the Euler angles.

Analyzing the dynamics involved in the physical device, the mathematical model can be derived according to Newton's law of motion like an ordinary aircraft. Therefore, the dynamic model of the multi-rotor, with *x*, *y,* and *z* as linear movements expressed as a result of roll (ϕ) or pitch (θ) rotations, can be obtained. Since the same motion laws apply for the different multi-rotor configurations, non-linear model design and its control can be applied to any multi-rotor configuration as they are obtained using the transformation and rotation matrices (1-1) and (1-2). The model is defined as follows [Arellano-Muro-13], [Bouabdallah-05], [Surya-16]

$$\begin{aligned} \ddot{\phi} &= \dot{\theta}\dot{\psi}\left(\frac{I_y - I_z}{I_x}\right) - \frac{J_r}{I_x}\dot{\theta}\Omega + \frac{l}{I_x}U_2 \\ \ddot{\theta} &= \dot{\phi}\dot{\psi}\left(\frac{I_z - I_x}{I_y}\right) + \frac{J_r}{I_y}\dot{\phi}\Omega + \frac{l}{I_y}U_3 \\ \ddot{\psi} &= \dot{\phi}\dot{\theta}\left(\frac{I_x - I_y}{I_z}\right) + \frac{l}{I_z}U_4 \\ \ddot{z} &= -g + (\cos\phi\cos\theta)\left(\frac{1}{m}\right)U_1 \\ \ddot{x} &= (\cos\phi\sin\theta\cos\psi + \sin\phi\sin\psi)\left(\frac{1}{m}\right)U_1 \\ \ddot{y} &= (\cos\phi\sin\theta\sin\psi - \sin\phi\cos\psi)\left(\frac{1}{m}\right)U_1 \end{aligned} \tag{1-3}$$

where I_x, I_y, and I_z are the moments of inertia of the multi-rotor frame body, J_r is the total inertia of the multi-rotor, Ω is the total angular speed combination generated by the motors, ϕ, θ, and ψ are the roll, pitch and yaw angles, g is the gravity acceleration, m is the total mass of the aircraft, x, y, and z are the translational linear movements of the vehicle, and U_1, U_2, U_3, and U_4 the control signals for the aircraft.

In order to obtain a state space model for system (1-3), the state vector system can be defined as follows

$$X = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \\ x_9 \\ x_{10} \\ x_{11} \\ x_{12} \end{bmatrix} = \begin{bmatrix} \phi \\ \dot{\phi} \\ \theta \\ \dot{\theta} \\ \psi \\ \dot{\psi} \\ z \\ \dot{z} \\ x \\ \dot{x} \\ y \\ \dot{y} \end{bmatrix}. \tag{1-4}$$

It can be observed that *x*, *y*, and *z*, which corresponds to the linear motion variables of the system, are directly affected by the control term U_1. The gyroscopic variables ϕ, θ and ψ are the results of the rigid rotation in space, dependent on U_2, U_3 and U_4, respectively.

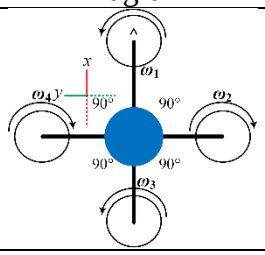
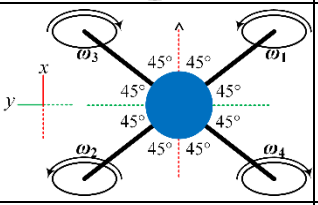
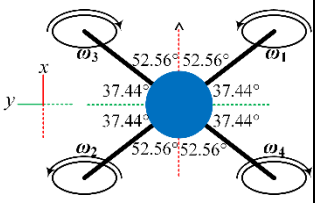
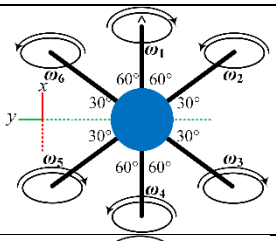
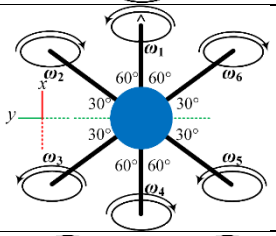
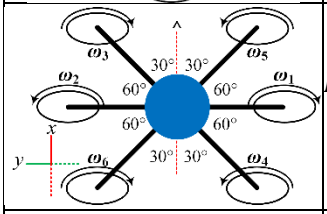
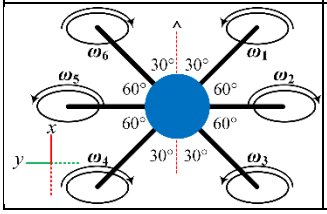
The relation between the control terms U_1, U_2, U_3 and U_4, and the angular velocities $\omega_1, \dots, \omega_n$ of the propellers of the multi-rotor depends on the configuration of the UAV, with n as the number of rotors of the vehicle. This relation is mathematically formalized with the following equation

$$U = M\Omega^2 \tag{1-5}$$

where M is the matrix of forces and moments generated by the motors. Table 1.1 shows some configurations considered in the literature or in commercial versions of quad-rotors and hexa-rotors, with their corresponding matrix of forces M and the arrangement of their motors. The vehicle's parameters related to matrix M are: C_T is the thrust coefficient, C_D is the drag coefficient, ρ is the air density, D is the propeller diameter, and A is the cross-section of the movement of the vehicle.

It is important to mention that each configuration is constituted of a set of rotors with a specific direction of rotation. A subset of the motors must rotate in clockwise (CW) $(+)$ direction and the rest in counterclockwise (CCW) $(-)$ direction to nullify the undesired overall gyroscopic moment on the vehicle, generated by the motors.

TABLE 1.1. MULTI-ROTORS MATRIX OF CONFIGURATIONS WITH MOTOR COMBINATIONS FOR QUAD/HEXA-ROTOR.

Diagram	Matrix of forces and motor configuration
	$M = \begin{bmatrix} b & b & b & b \\ 0 & b & 0 & -b \\ b & 0 & -b & 0 \\ d & -d & d & -d \end{bmatrix}, \begin{matrix} b = C_T \rho D^4 \\ d = C_D \rho A/2 \end{matrix}$ $\Omega = -\omega_1 + \omega_2 - \omega_3 + \omega_4$ [Bouabdallah-05]
	$M = \begin{bmatrix} b & b & b & b \\ p & -p & -p & p \\ p & -p & p & -p \\ d & d & -d & -d \end{bmatrix}, \begin{matrix} b = C_T \rho D^4 \\ p = b\sin(45°) = b\sqrt{2}/2 \\ d = C_D \rho A/2 \end{matrix}$ $\Omega = -\omega_1 - \omega_2 + \omega_3 + \omega_4$ [ADT-16a]
	$M = \begin{bmatrix} b & b & b & b \\ p & -p & -p & p \\ q & -q & q & -q \\ d & d & -d & -d \end{bmatrix}, \begin{matrix} b = C_T \rho D^4 \\ p = b\sin(52.56°) \\ q = b\sin(37.44°) \\ d = C_D \rho A/2 \end{matrix}$ $\Omega = -\omega_1 - \omega_2 + \omega_3 + \omega_4$
	$M = \begin{bmatrix} b & b & b & b & b & b \\ 0 & p & p & 0 & -p & -p \\ b & q & -q & -b & -q & q \\ -d & d & -d & d & -d & d \end{bmatrix}, \begin{matrix} b = C_T \rho D^4 \\ p = b\sin(60°) = b\sqrt{3}/2 \\ q = b\sin(30°) = b/2 \\ d = C_D \rho A/2 \end{matrix}$ $\Omega = \omega_1 - \omega_2 + \omega_3 - \omega_4 + \omega_5 - \omega_6$ [Baránek-12]
	$M = \begin{bmatrix} b & b & b & b & b & b \\ 0 & -p & -p & 0 & p & p \\ b & q & -q & -b & -q & q \\ d & -d & d & -d & d & -d \end{bmatrix}, \begin{matrix} b = C_T \rho D^4 \\ p = b\sin(60°) = b\sqrt{3}/2 \\ q = b\sin(30°) = b/2 \\ d = C_D \rho A/2 \end{matrix}$ $\Omega = -\omega_1 + \omega_2 - \omega_3 + \omega_4 - \omega_5 + \omega_6$ [Arellano-Muro-13]
	$M = \begin{bmatrix} b & b & b & b & b & b \\ b & -b & -q & q & q & -q \\ 0 & 0 & p & -p & p & -p \\ -d & d & -d & d & d & -d \end{bmatrix}, \begin{matrix} b = C_T \rho D^4 \\ p = b\sin(60°) = b\sqrt{3}/2 \\ q = b\sin(30°) = b/2 \\ d = C_D \rho A/2 \end{matrix}$ $\Omega = \omega_1 - \omega_2 + \omega_3 - \omega_4 - \omega_5 + \omega_6$ [ADT-16a]
	$M = \begin{bmatrix} b & b & b & b & b & b \\ q & b & q & -q & -b & -q \\ p & 0 & -p & -p & 0 & p \\ d & -d & d & -d & d & -d \end{bmatrix}, \begin{matrix} b = C_T \rho D^4 \\ p = b\sin(60°) = b\sqrt{3}/2 \\ q = b\sin(30°) = b/2 \\ d = C_D \rho A/2 \end{matrix}$ $\Omega = -\omega_1 + \omega_2 - \omega_3 + \omega_4 - \omega_5 + \omega_6$ [Surya-16]

The sign of each coefficient of M is obtained from the analysis of the CW and CCW motor rotations when placed at each of the coordinate axis. The analysis depends on the orientation defined for the Cartesian plane in the UAV, and on the IMU sensor orientation in the controller board. For instance, the three linear movements and three rotations of the aircraft can be obtained following Fig. 1.2. Roll is generated by increasing the speed in motors 1 and 3 and decreasing it in 2 and 4, or vice versa, to get the other direction. Pitch is obtained by increasing motor speed in 2 and 3 and decreasing it in 1 and 4, or vice versa, to get the other direction. Yaw is obtained by increasing the speed in 1 and 2 and decreasing it in 3 and 4, or vice versa. It is worth to note that this motion generation corresponds exclusively to the motor and frame configuration shown in the figure.

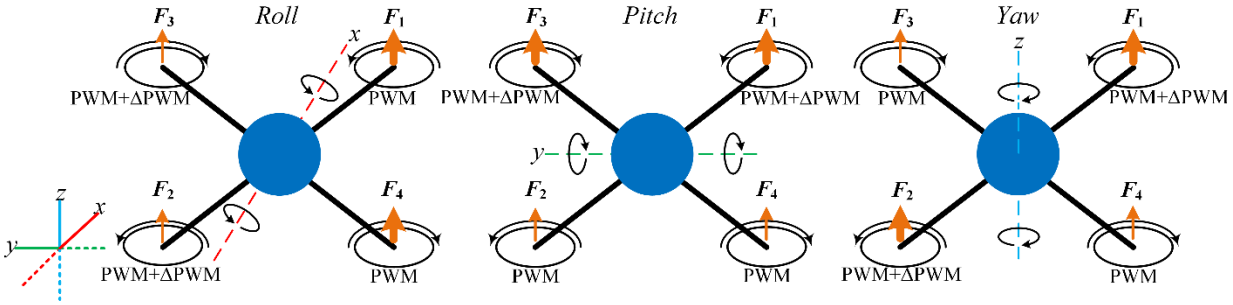


Fig. 1.2 Linear and rotational motion generation for a quad-rotor.

From a control design point of view, equation (1-5) can be used to relate the output of the actuators ($\omega_1, \dots, \omega_n$) to the inputs of the vehicle (U_1, U_2, U_3 and U_4). Hence, to determine the references for the actuators, the term Ω must be solved from which results of the form

$$\Omega = \sqrt{M^{-1}U}. \tag{1-6}$$

For the cases where matrix M is not square, it is impossible to obtain a unique solution for the system as it is composed of fewer unknowns than equations. Nonetheless, a workaround to obtain a suitable solution can be implemented by means of the Moore-Penrose pseudo-inverse. Considering a non-square m (rows) by n (columns) matrix, and if $m \geq n$, then $M^+ = (M^T M)^{-1} M^T$ satisfies the pseudo-inverse, if $m \leq n$, then $M^+ = M^T (M M^T)^{-1}$ satisfies the pseudo-inverse [MacAusland-14].

In order to obtain a more concise expression for the dynamical model of the system, the following equalities are defined

$$a_1 = \frac{I_y - I_z}{I_x}, a_2 = -\frac{J_r}{I_x}, a_3 = \frac{I_z - I_x}{I_y}, a_4 = \frac{J_r}{I_y}, a_5 = \frac{I_x - I_y}{I_z} \tag{1-7}$$

$$b_2 = \frac{l}{I_x}, b_4 = \frac{l}{I_y}, b_6 = \frac{l}{I_z}, b_8 = (\cos x_1 \cos x_3)\left(\frac{1}{m}\right) \tag{1-8}$$

$$f_2 = x_4 x_6 a_1 + x_4 a_2 \Omega, \qquad f_4 = x_2 x_6 a_3 + x_2 a_4 \Omega, \qquad f_6 = x_4 x_2 a_5, \qquad f_8 = -g \tag{1-9}$$

$$\begin{aligned} u_x &= (\cos x_1 \sin x_3 \cos x_5 + \sin x_1 \sin x_5) \\ u_y &= (\cos x_1 \sin x_3 \sin x_5 - \sin x_1 \cos x_5) \end{aligned} \tag{1-10}$$

which, in combination with (1-3) and (1-4), results in a simplified state-space model given by [Bouabdallah-05]

$$\dot{X} = f(X, U) = \begin{bmatrix} x_2 \\ f_2 + b_2 U_2 \\ x_4 \\ f_4 + b_4 U_3 \\ x_6 \\ f_6 + b_6 U_4 \\ x_8 \\ f_8 + b_8 U_1 \\ x_{10} \\ u_x\left(\frac{1}{m}\right) U_1 \\ x_{12} \\ u_y\left(\frac{1}{m}\right) U_1 \end{bmatrix}. \tag{1-11}$$

At this point, a complete dynamical system for the vehicle is defined and the controller design process can be developed. Depending on the flight mode to execute in the vehicle, the variables to control are selected. In this work, trajectory and stabilize flight mode controllers are deployed using different control approaches. [Fan-17], [Duc-15].

1.2. Dynamical Model of the Actuators

The motors attached to the aircraft are brushless direct current (BLDC) electric motors driven by electronic speed controllers (ESC). This kind of motors come in one, two, or three-phase configuration, being the latter the most common version [Keeping-13] and the one used during the development of the presented work. In order to actuate these motors, the ESC generates commutations to sequentially energize the stator coils, generating a rotating electric field that interacts with N pairs of magnet pairs mounted on the rotor. The result is a torque that drags the rotor around with the electric field, generating the angular velocity of the rotor. Thus, *N* electrical revolutions equate to one mechanical revolution.

In the case of the three-phase motors, three Hall-effect sensors are commonly embedded in the stator to indicate the relative positions of stator and rotor to the controller, so it can energize the windings in the correct sequence and at the correct time. The Hall-effect sensors are usually mounted on the non-driving end of the unit, as shown in Fig. 1.3.

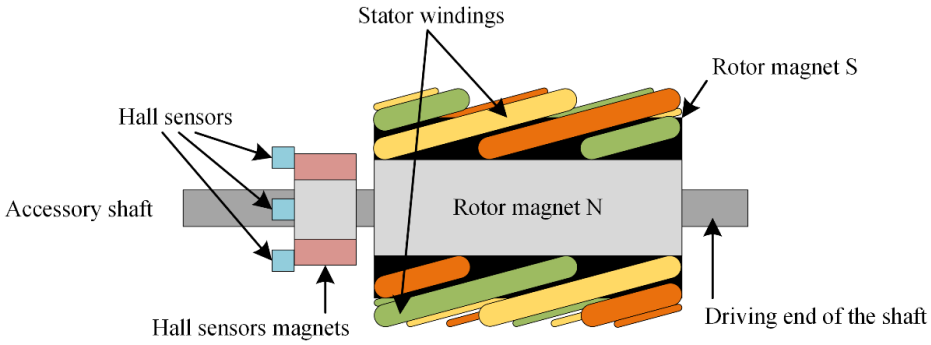


Fig. 1.3 Brushless DC motor components diagram.

All electric motors generate a voltage potential due to the movement of the windings through the associated magnetic field. This potential is known as an electromotive force (EMF) and it gives rise to a current in the windings with a magnetic field that opposes the original change in the magnetic flux; in other words, EMF tends to resist the rotation of the motor and is therefore referred to as back EMF [Keeping-13]. By monitoring the back EMF or the Hall sensors, and by using a microcontroller for that purpose, the relative positions of the stator and rotor can be determined. However, the motors being used for this thesis do not have Hall sensors, instead, encoders with reflective sensors are sampled altogether with current sensor obtain the rotor speed and the power that each ESC consumes. That said, the next step in the modeling is to determine the DC-motor equations that relate the angular speed of the rotor, voltage and current in the stator as follows ([Bouabdallah-05], [Utkin-99] and [Kumpanya-15])

$$V_a(t) = R_a i_a(t) + L_m \frac{di_a(t)}{dt} + V_e(t) \tag{1-12}$$

$$\tau_m(t) = J_m \frac{d\omega_m(t)}{dt} + B_m \omega_m(t) + \tau_l(t) \tag{1-13}$$

$$\tau_m(t) = k_\tau i_a(t) \tag{1-14}$$

$$V_e(t) = k_e \omega_m(t) \tag{1-15}$$

where $V_a(t)$ is armature voltage, R_a is the motor armature internal resistance, $i_a(t)$ is the motor armature current, L_m is the motor inductance, $V_e(t)$ is the back EMF voltage, $\tau_m(t)$ is the motor

torque, J_m is the moment of inertia seen by the motor, B_m is the coefficient of viscous friction, $\omega_m(t)$ is the motor angular speed, τ_l is the motor load, k_τ is the torque constant and k_e is the back EMF constant.

To simplify the process, the equations (1-12) and (1-13) are rewritten as follows in order to solve the current and angular speed derivatives

$$L_m \frac{di(t)}{dt} = V_a(t) - R_a i_a(t) - k_e \omega_m(t) \tag{1-16}$$

$$J_m \frac{d\omega_m(t)}{dt} = k_\tau i_a(t) - B_m \omega_m(t) - \tau_l(t). \tag{1-17}$$

Previously it was mentioned that the BLDC motors used in this dissertation are three-phase actuators, but aiming to simplify the computation of the math and the model, an electrical approximation from a three-phase to a direct current (DC) motor model is proposed, as shown in Fig. 1.4. It is important to note that only the current and torque mathematical models are being considered, for the simplification into a DC motor model that this work is pursuing.

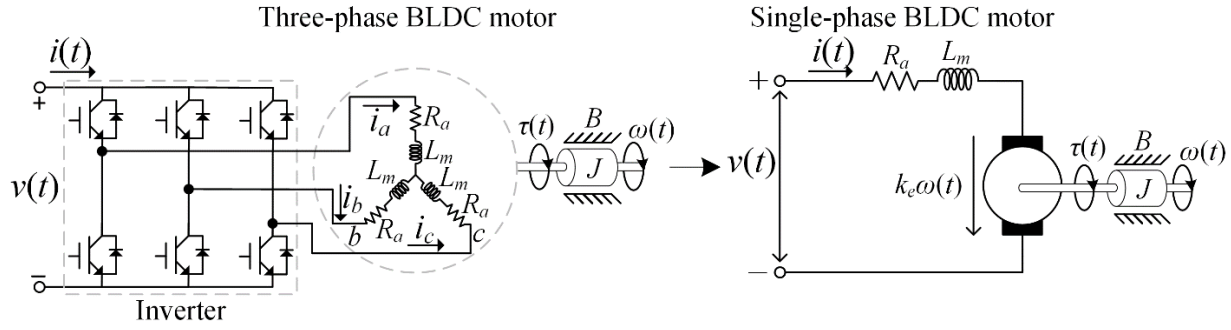


Fig. 1.4 Three-phase BLDC to single-phase BLDC brushless motor electric diagram.

1.3. Flight Modes for Multi-Rotors

The flight mode of a UAV determines the way that the embedded controller of the vehicle interprets the commands received from user by means of a remote control. It also determines the structure of the controller as the flight mode defines which variables of the vehicle will be controlled and monitored. The platform and software that is being used in the development of this work is the Pixhawk and the ArduPilot project, which supports a number of 20 possible flight modes that are listed in Table 1.2 [ADT-18b]. For the purpose of this dissertation, two flight modes are considered: the stabilize flight mode which controls the attitude and altitude of the multi-rotor,

and the trajectory tracking flight mode which controls the position and yaw angle of the aircraft.

TABLE 1.2. MULTI-ROTOR FLIGHT MODES SUPPORTED BY THE PIXHAWK AND ARDUPILOT PROJECT.

Flight mode	**Altitude control**	**Position control**	**GPS dependency**	**Description**
Acrobatic	Manual control	Manual control	No	Holds attitude, no self-level.
Altitude-Hold	Automated stabilize control	Manual control with limits and self-level	No	Holds altitude and self-levels the roll and pitch angle.
Auto	Automatic control	Automatic control	Yes	Executes pre-defined mission
Auto-Tune	Automated stabilize control	Automatic control	Yes	Automated pitch and tilt procedure to improve control gains applied in the control algorithm.
Brake	Automated stabilize control	Automatic control	Yes	Brings copter to an immediate stop.
Circle	Automated stabilize control	Automatic control	Yes	Automatically circles a point in front of the vehicle.
Drift	Manual control	Manual control with limits and self-level	Yes	Self-levels the roll and pitch axis, but coordinates yaw with roll like a plane.
Flip	Automatic control	Automatic control	No	Rises and completes an automated flip.
Follow	Automated stabilize control	Automatic control	Yes	Follows a GPS on the ground.
Guided	Automatic control	Automatic control	Yes	Navigates to single points commanded by the ground control station (GCS).
Land	Automatic control	Automated stabilize control	No	Reduces altitude to ground level, attempts to go straight down.
Loiter	Automated stabilize control	Automated stabilize control	Yes	Holds altitude and position, uses GPS for movements.
Position-Hold	Automated stabilize control	Manual control with limits and self-level	Yes	Holds altitude and position, uses GPS for movements, but manual roll and pitch when the RC sticks are not centered.
Return to Launch (RTL)	Automatic control	Automatic control	Yes	Returns to the take-off location and may also include the landing feature.
Simple/Super Simple	N/A	N/A	Yes	An add-on to flight modes to use the user's view instead of yaw orientation.
Smart RTL	Automatic control	Automatic control	Yes	RTL, but traces path to get the home location.
Stabilize	Manual control	Manual control with limits and self-level	No	Self-levels the roll and pitch axis.
Sport	Automated stabilize control	Automated stabilize control	No	Alt-hold, but holds pitch and roll when the RC sticks are centered
Throw	Automatic control	Automatic control	Yes	Holds position after a throwing take-off.

1.4. Control Schemes

As mentioned in the previous sub-section, the flight mode chosen for the aerial vehicle defines the control scheme that must be applied to maneuver the vehicle. In that sense, several

control algorithms as linear quadratic regulator/Gaussian (LQR/LQG), adaptive control, robust control, optimal control, feedback linearization, and intelligent control (fuzzy logic and artificial intelligence) have been applied to multi-rotor models to achieve stabilization and trajectory tracking. All these can be categorized as linear or non-linear, each of them having advantages and disadvantages for controlling a multi-rotor. Next, the characteristics of proportional integral derivative (PID) control, the backstepping technique, and sliding mode controllers are presented as they are commonly used in UAV devices [Zulu-14] and will be used throughout the development of the presented work.

1.4.1 Proportional-Integral-Derivative (PID)

This is the one of the most common controllers used in the industry due to its advantage of having only three control gains that can be easily adjusted, and due to the relatively good robustness provided. Nevertheless, the multi-rotor mathematical model implies non-linearities and an imprecise nature due to un-modeled dynamics, causing the PID to have performance limitations. Moreover, the tuning could present some challenges as it must be conducted experimentally.

Due to its simplicity, this controller does not guarantee optimal control in many situations. One of the reasons is that it is not strictly necessary to know the mathematical model of the plant as it has demonstrated that can provide satisfactory control under limited circumstances.

PIDs are based on three gains applied to the error, and these are the proportional gain (K_P), the integral (K_I), and the derivative (K_D). The error calculation is the basis to control the UAV, as shown in Fig. 3.1. Such error is the difference between the desired value (reference) for the controlled the variable and its actual measured value (attitude or position). For example, when controlling the roll angle, the desired reference is defined and then the measured value in the accelerometer sensor is subtracted to get the error signal that feeds the PID controller. The error is computed by the control scheme obtaining control signals that are later translated into PWMs to inject into the ESC devices to actuate the motors. The traditional PID controller parallel implementation is defined by

$$U(t) = K_P e(t) + K_I \int_0^t e(t)\, dt + K_D \frac{de(t)}{dt} \tag{1-18}$$

where $e(t)$ is the error variable, K_P, K_I and K_D are the proportional, integral and derivative control

gains, and $U(t)$ is the resulting control signal [Duc-15].

In order to define the values of the controller gains, tuning methods are available to provide a starting point for the desired behavior in the system, as the Ziegler-Nichols methods that base the settings in the delay time L and the time constant T of the step response of the system, or the method of defining a critical value K_{cr} where the system presents sustained oscillations and the period P_{cr} is obtained. Based on these values, K_P, K_I and K_D are set with well-known formulas [Ogata-96]. There are other multiple methods for controller tuning like the frequency response design or the computational optimization algorithm, but the purpose of this work is not to provide a detailed explanation of their functionality.

Other approaches can be followed for obtaining the controller values, like the effects produced in the closed-loop system when changing each of the parameters, as shown in Table 1.3. The modification of each parameter is not independent, meaning that the modification of each of them can affect the impact of the others, and for that reason, the table should only be used as a reference [UM-99].

TABLE 1.3. PID CONTROL GAINS EFFECTS ON THE CLOSED-LOOP SYSTEM TIME RESPONSE.

Control gain	Rise time	Overshoot	Settling time	Steady-state error
K_P	Decrease	Increase	Small change	Decrease
K_I	Decrease	Increase	Increase	Eliminate
K_D	Small change	Decrease	Decrease	No change

1.4.2 Backstepping Control

The backstepping technique is a recursive Lyapunov-based feedback control scheme and the idea behind it is to design a controller recursively by considering some state variables as virtual controls, to design intermediate control laws that guarantee the exponential convergence of the error to zero, and the global stabilization of the system [Wang-09]. This is achieved by breaking down the controller into steps and progressively stabilizing each subsystem.

The advantages of this algorithm are the convergence speed that does not require a lot of computational resources, and disturbances handling, but not being one of the most robust methodologies. For increasing robustness, an integrator is commonly added, and that eliminates the steady-state errors of the system, reduces the response time, and restrain overshoot of control

parameters. Another option is to include a discontinuous term in the control signal based on sliding mode control or another variable structure control approach.

1.4.3 Sliding Mode Control (SMC)

This is a non-linear control algorithm that works by applying discontinuous control signals to the system to command it to slide along a prescribed surface of the state plane of the system. One of its main advantages is that it does not simplify the dynamics through linearization and offers good tracking capabilities. It has proved robustness by being tested with full-actuated and underactuated systems, showing good stability. The disadvantages of this method are the oscillations of finite frequency, commonly known as chattering, that are present when trying to reach the zero error due to the discontinuous nature of the control law.

The term sliding mode control first appeared in the context of relay systems, and the reason is that the control as a function of the system state switches at high frequency; this motion is called sliding mode. This type of controller played and is still playing an exceptional role in theoretical and practical applications due to its order reduction property and its low sensitivity to disturbances and plant parameter variations, which makes it an effective tool for controlling complex high-order dynamic plants. [Utkin-99]

For instance, considers the following nonlinear system

$$\dot{x} = f(x) + u, \quad y = Cx \tag{1-19}$$

with $f(x)$ as a manifold of the states vector x which is bounded by $|f(x)| < f_0 = c$, where c is a constant; y is the output of the system defined as a linear combination of the states given by the output matrix C; and u is the control term of the system. If the error variable is defined as $e = r(t) - y$ where $r(t)$ is the output reference, then, a sliding mode controller can be designed as a relay function of the tracking error as follows

$$u = \begin{cases} u_0 \text{ if } e > 0 \\ -u_0 \text{ if } e < 0 \end{cases} \text{ or } u = u_0 \text{sign}(e), u_0 > c. \tag{1-20}$$

A graphical representation of the sign function of the error is depicted in Fig. 1.5.

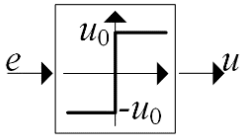


Fig. 1.5 Graphical representation of the sign function.

The values of e and $\dot{e} = \dot{r} - f(x) - u_0 \text{sign}(e)$ are opposite if $u_0 > f_0 + |\dot{r}|$, meaning that the magnitude of the error decreases at a finite rate and the error is equal to zero after the finite interval T as shown in Fig. 1.6. The motion occurring at $t > T$ is called sliding mode.

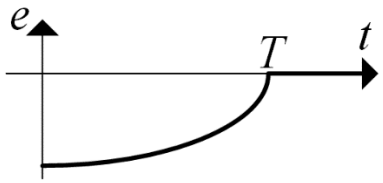


Fig. 1.6 Error variable behavior during the sliding mode in a closed-loop system.

For a real-life implementation, bandwidth limits in the switching device generates high frequency components in the closed loop behavior of the system. Also, the high frequency switching commands sent to the actuators could led to their premature wearing or damage.

1.5. Control Algorithms Comparison

Table 1.4 [Zulu-14] summarizes various algorithms discussed above when applied to multi-rotors under the same circumstances. It is not an elaborate study of the different control options rather than a practical guide in accordance with each controllers' capabilities, where the legends values are defined as follows: 0 stands for low, 1 stands for average, and 2 means high. The algorithms 1 through 5 correspond to linear controllers, while the remaining to non-linear controllers.

1.6. Hardware and Software for UAV's Embedded Controller

The main focus of this work is on the multi-rotor area, and with that aim, a list of embedded software and hardware solutions for the implementation of controller on these aerial vehicles is defined. In Table 1.5, Table 1.6, Table 1.7, Table 1.8, Table 1.9, and Table 1.10 the most common flight control boards and their driving firmware are listed, along with a succinct description of their capabilities. Notice that these lists are not definitive, and many other platforms and firmware may be available.

TABLE 1.4. COMPARISON OF MULTI-ROTOR CONTROL ALGORITHMS.

Control algorithm	**Characteristics**												
	Robust	Adaptive	Optimal	Intelligent	Tracking ability	Fast convergence/response	Precision	Simplicity	Disturbance rejection	Un-modeled parameter handling	Manual tuning	Signal noise	Chattering/energy loss
1. PID	1	0	0	0	1	1	1	2	0	0	2	2	0
2. Intelligent PID	1	0	0	2	1	1	1	1	0	0	0	1	0
3. LQR	0	2	1	0	1	1	0	1	1	0	1	1	0
4. LQG	0	2	2	0	1	1	0	0	2	0	1	0	0
5. L_1	0	2	2	0	1	2	2	0	1	0	0	0	0
6. H_∞	2	1	2	0	2	0	1	0	1	1	0	0	0
7. SMC	1	2	1	0	2	2	2	1	2	1	0	0	2
8. FBL	1	1	0	0	2	2	2	1	1	1	0	1	2
9. Backstepping	0	2	0	0	2	0	1	0	2	1	0	0	0
10. Fuzzy logic	1	1	1	2	1	1	1	1	1	0	1	0	0
11. Neural networks	1	2	2	2	1	1	1	0	1	1	0	0	0
12. Genetic	1	2	2	2	1	1	1	0	1	2	0	0	0

TABLE 1.5. CLEANFLIGHT FIRMWARE AND COMMON HARDWARE PLATFORMS USED FOR MULTI-ROTOR UAV CONTROL.

Platform	**Hardware description**	**Software description**
Naze32	STM32F103 microcontroller based board which contains MPU6500 IMU, barometer, and can handle up to 2 UARTs and 6 pulse protocol outputs.	Open source code designed for acrobatic and racing drones. It supports STM32F1/3/4/7 microcontroller based boards. Multiple telemetry protocols support as FrSky, SmartPort, S. Port, HoTT, iBus, MavLink, CRSF, and SRXL [Cleanflight-18].
Seriously Dodo Flight controller	STM32F303 microcontroller based board which contains MPU6050 IMU, barometer, 3 UART ports available and can handle up 8 pulse protocol outputs [RMRC-18].	

TABLE 1.6. ROBOT OPERATING SYSTEM (ROS) FIRMWARE AND COMMON HARDWARE PLATFORMS USED FOR MULTI-ROTOR UAV CONTROL.

Platform	**Hardware description**	**Software description**
Multiple boards	N/A.	An open-source operating system designed for robotics development that provides libraries and tools for hardware abstraction, device drivers, visualizers, message-passing, localization, mapping, navigation, etc. [ROS-18].

TABLE 1.7. ARDUPILOT FIRMWARE AND COMMON HARDWARE PLATFORMS USED FOR MULTI-ROTOR UAV CONTROL.

Platform	**Hardware description**	**Software description**
ArduPilot[1] Mega (APM)	ATMEGA2560 microcontroller based board that has an MPU6000 IMU, an ATMEGA32U2 for PPM encoding and USB interfacing, 2 serial ports available, I^2C port, SPI port, 12 ADC channel ports, barometer, magnetometers, and 8 PWM outputs [ADT-18a].	Open source multi-platform code. Supports multiple commands modes depending on the vehicle needs, provides user-specific scheduler routines for easy development, and several numbers of libraries. It provides advanced failsafe and video capabilities to the user. Full data logging is available and real-time communication. Multiple setup configurations are provided for specific needs. Different types for RC inputs are supported, such as PPM Sum, S. Bus, DSM, PWM, and MAVLink. It supports I^2C, SPI, UART, and CAN protocols [ADT-18b].
Pixhawk[2]	STM32F427 microcontroller based board that uses an STM32F100 as a failsafe system. It has an L3GD20H gyroscope, LSM303D and MPU6000 IMU, and an MS5611 barometer. It provides 1 ADC channel, 1 CAN port, 1 I^2C port, 5 UART ports, and supports up to 14 pulse protocol outputs [PX4DT-18].	
Navio2/ Raspberry	Daughter I/O board that provides the same capabilities as the Pixhawk but with an integrated GNSS receiver. It has connectivity ports for Raspberry for running the ArduPilot firmware in Linux [EMLID-18].	
Erle-Brain 3	ARM Cortex-A53 CPU based board that runs Linux operating system and ArduPilot code. It incorporates a gravity sensor, gyroscope, digital compass, pressure sensor, temperature sensor, 12 pulse protocol outputs and ADC for battery sense. It provides 1 serial port, 2 I^2C ports, built-in Wi-Fi, Bluetooth, Ethernet, 4 USB ports and can handle an 8MP camera [Erle-Robotics-18a].	
PXFMINI/ Raspberry	This is a daughter I/O board that provides the same sensing capabilities as the Erle-Brain 3, but only 8 pulse protocol outputs, and a communication port for being connected to a Raspberry for running the ArduPilot firmware in Linux [Erle-Robotics-18b].	
Qualcomm Snapdragon Flight	Krait quad-core based board with a DSP for real-time control. It has built-in 2G/5G Wi-Fi, Bluetooth, USB, and GNSS. It provides an integrated 4K camera, IMU, barometer, ESX connector, and an expansion port connector. It runs Linux on the main processor and ArduPilot code in the DSP [Qualcomm-18].	
Intel Aero Compute	Intel Atom x7-Z8750 based board with Wi-Fi connectivity, USB connector, USB 3.0 dedicated for Intel RealSense support. It provides 3 UART ports, 1 I^2C port, and 1 ADC. It supports an I/O expansion board and runs Linux and ArduPilot code.	

TABLE 1.8. LIBREPILOT FIRMWARE AND COMMON HARDWARE PLATFORMS USED FOR MULTI-ROTOR UAV CONTROL.

Platform	**Hardware description**	**Software description**
OpenPilot CopterControl 3D (CC3D)	STM32F303 microcontroller based board which contains MPU6050 IMU, 1 serial port available and can handle up to 10 PWM outputs [LibrePilot-18a].	Open source code project for multi-rotors and other RC devices. It supports multiple inputs protocols as PWM, PPM, S. Bus, DSM, Sat, SRLX, HoTT, EX Bus, OpenLRS, and iBus, and multiple boards [LibrePilot-18b].

[1] ArduPilot, Version 3.5.7, jDrones & Co. 9 Chaloem Phrakiat Rd., Soi 83, Bangkok 10250, Thailand, 2018.
[2] Pixhawk, Version 2.4.6, Auterion, Giesshuebelstrasse 40, 8045 Zurich, Switzerland, 2018.

TABLE 1.9. SAFESMART FLIGHT CONTROL FIRMWARE AND COMMON HARDWARE PLATFORMS USED FOR MULTI-ROTOR UAV CONTROL.

Platform	Hardware description	Software description
SafeSmart Autopilot	Microchip dsPIC33 microcontroller-based platform that has an accelerometer, gyroscope, magnetometer, barometer, and GPS sensor. It has 4 UART ports, supports CAN protocol, I2C, and SPI. For motor control, it provides up to 8 pulse protocol channels.	Toolbox for Matlab[3] that offers control capabilities and enables the user to generate the code for the platform and reprogram it [IntelinAir-18].

TABLE 1.10. FLYTOS FIRMWARE AND COMMON HARDWARE PLATFORMS USED FOR MULTI-ROTOR UAV CONTROL.

Platform	Hardware description	Software description
Multiple boards	N/A.	Operating system built on ROS and Linux that works with ArduPilot based controllers, NVIDIA TX1/TX2, ODROID XU4, Raspberry, DJI, and Intel Aero Compute. It provides 4G/LTE capabilities, video streaming, swarm managing, object tracking, indoor navigation, thermal sensing, LIDAR sensing, collision avoidance features, and 3D motion tracking [FlytBase-18].

1.7. Conclusions

Numerous types of UAVs can be found in the state of the art of these devices, commercially and for research purposes. Nevertheless, VTOL aircrafts have become popular due to the simplified launching mechanisms or successful results under non-controlled environments for mission execution. As mentioned, different control algorithms can be used for flying controllability purposes, being PID one of the most used due to the simplicity and lack of a mathematical model. This control is not robust for situations where disturbances are present, in that case, a more specialized mechanism is required, such as backstepping or sliding mode; nevertheless, such algorithms require numerous aircraft parameters that are not commonly specified and demand characterization.

As a variety of control techniques are available, the same pattern is found when talking about hardware and software for UAV handling. As technology and complexity evolve, additional features are needed in the UAVs for providing reliability, repeatability, and enough processing capabilities while maintaining or decreasing the power consumption, that is the reason why an

[3] MATLAB, Version 2017b, The MathWorks, Inc., 3 Apple Hill Drive, Natick, MA 01760-2098, 2017.

enormous amount of options is available.

In the next chapters, some of the control technologies are discussed and backstepping is taken for embedded implementation and real-time execution. Code development and debug are performed for both simulation real-time environments in order to reproduce the results and achieve the goal of this dissertation.

2. UAV Parameters Characterization and Dynamical Modeling

Aiming to control the UAV, the first step in the process is to define the variables of interest of the system and express their relationship using a dynamic model. This process must be done for the vehicle and its actuators, accordingly to the proposed overall control scheme. Once the mathematical model is defined, the parameters of the model must be characterized by means of real time experiments or obtained through the datasheet from the manufacturer. Another option is to obtain a simple input-output dynamical model, as a transfer function, from experimental data of the system.

The latter method is utilized in this work as transfer functions for the motor speed and current consumption of the actuators were obtained using data from real time experiments consisting on step inputs to the motor and acquiring the output time response. The information provided by the response to the step impulse allows the usage of the state variables representation, and at the same time, discloses most of the physical parameters involved in the system. Such characterization is required for feeding the controller with real values to achieve the maneuverability of the aircraft, and for reducing controllability errors once the control is implemented in the real UAV platform.

Furthermore, nominal values for the dynamical parameters of the vehicle are obtained using a 3D CAD model of the vehicle which replicates almost exactly the physical properties of the real UAV. The process to develop these characterizations is described in the following subsections.

2.1. Actuators Dynamics Modeling

As mentioned before, brushless DC electric motors are used to actuate the UAV of the presented work. These actuators are in the three-phase configuration being controlled by electronic speed controllers. This research pursues a simplified model of the actuator by considering them as single-phase devices, from the motor model perspective standpoint. According to that, the transfer function is a suitable model to be obtained by stimulating the system with a step input signal. This

section presents the procedure to estimate the parameters of the simplified model of the actuators.

2.1.1 Current and Speed Transfer Function Identification

In order to obtain the motor voltage to speed transfer function, the step response for the motor is required. For that purpose, the ESC is injected with a 100 percent duty cycle for a time-lapse and later decreased to 50 percent duty cycle to observe the transient response. The current consumption and speed responses to the step signal are displayed in Fig. 2.1 and Fig. 2.2, respectively.

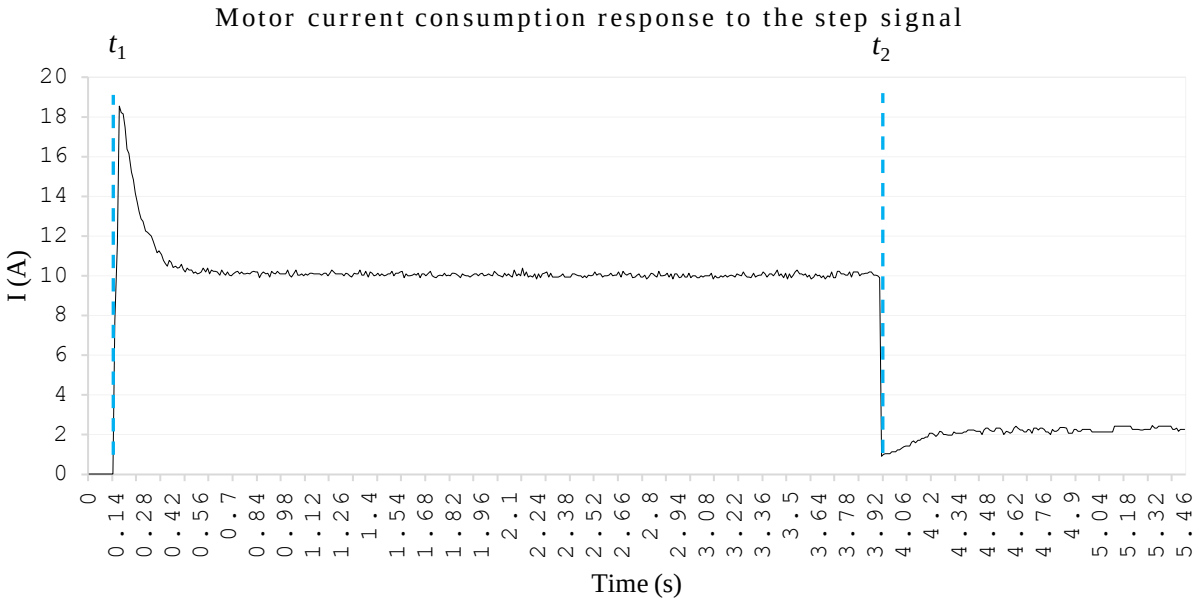


Fig. 2.1 Motor current consumption response to step signal and transient operation.

It can be noticed that the speed signal does not have the first overdamping that is seen in the current i response, and that is because the motors can demand more current when breaking the static motion to increase their speed, but the velocity capabilities cannot be exceeded from the specifications.

Once all the information is generated, the data is imported to the Matlab system identification tool, where the number of system poles and zeros are set as desired for calculating the transfer function model. In that regard, a number of 100 system identification iterations were executed for each combination, starting from 2 poles and 0 zeros, up to 5 poles 5 zeros, having the best accuracy combinations for current and speed listed in Table 2.1 and Table 2.2,

correspondingly.

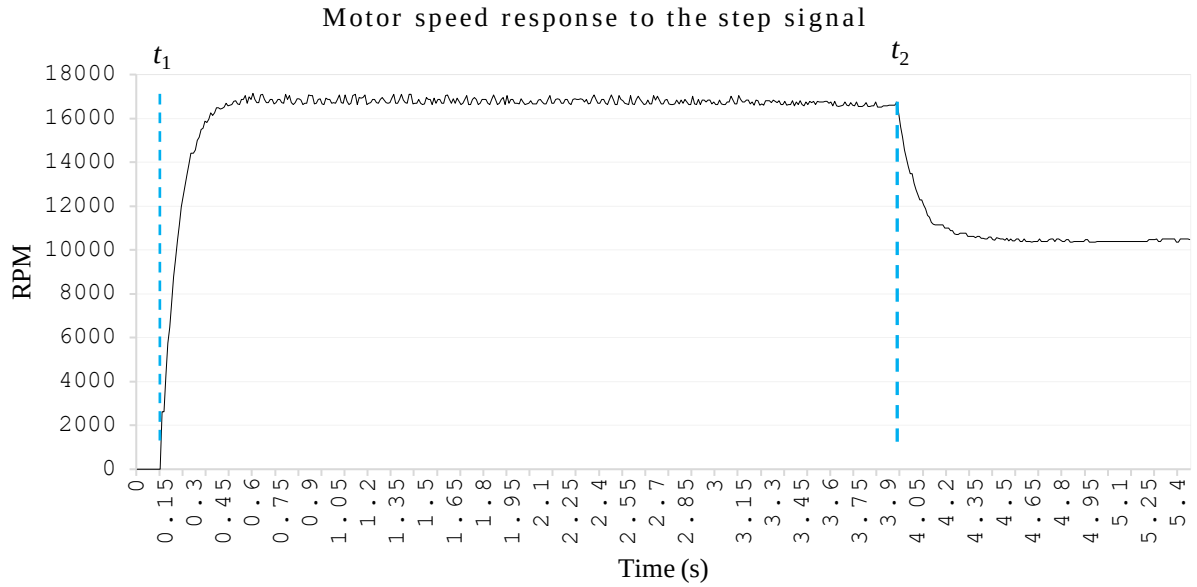


Fig. 2.2 Motor speed response to the step signal and transient operation.

TABLE 2.1. TRANSFER FUNCTIONS FOR THE PWM DUTY CYCLE TO CURRENT CONSUMPTION, DIFFERENT POLES AND ZEROS COMBINATIONS IN ACCURACY ORDER.

Poles	Zeros	Transfer function obtained	Accuracy
4	4	$\frac{10.33s^4 + 14.74s^3 + 3.332s^2 + 0.1483s + 0.0001406}{s^4 + 0.7339s^3 + 0.1905s^2 + 0.008996s + 1.414 \times 10^{-5}}$	92.65%
3	3	$\frac{11.27s^3 + 12.97s^2 + 0.8405s - 8.744 \times 10^{-5}}{s^3 + 0.6198s^2 + 0.05683s + 9.977 \times 10^{-6}}$	89.38%
5	3	$\frac{339.5s^3 + 58.44s^2 + 3.497s + 0.002232}{s^5 + 21.57s^4 + 22.26s^3 + 4.21s^2 + 0.2338s + 0.0002603}$	84.81%
3	2	$\frac{21.12s^2 + 1.351s - 0.0004505}{s^3 + 1.09s^2 + 0.09735s + 8.211 \times 10^{-10}}$	82.2%
5	2	$\frac{52.99s^2 + 2.532s - 0.0003044}{s^5 + 3.734s^4 + 6.3s^3 + 3.178s^2 + 0.1761s + 5.579 \times 10^{-12}}$	76.73%
3	1	$\frac{40.39s + 0.003151}{s^3 + 1.309s^2 + 3.142s + 0.002591}$	73.46%
2	2	$\frac{16.48s^2 + 0.8366s + 0.002468}{s^2 + 0.05476s + 0.0003465}$	59.63%
2	0	$\frac{24.05}{s^2 + 0.8608s + 2.477}$	52.71%

In the simulation environment, the current response transfer function of 4 poles and 4 zeros equation is chosen from the list of transfer functions, and for the speed response the 2 poles and 2 zeros, because of its performance and best fit for execution matching. In the real world, without considering perturbations as airspeed, friction, or other system parameters, the DC motor transfer function is defined with 2 poles and 0 zeros, and this needs to be used for obtaining the motor

specifications.

TABLE 2.2. TRANSFER FUNCTIONS FOR THE PWM DUTY CYCLE TO MOTOR RPM SPEED, DIFFERENT POLES, AND ZEROS COMBINATIONS IN ACCURACY ORDER.

Poles	Zeros	Transfer function obtained	Accuracy
2	2	$\frac{716.6s^2 + 1.377 \times 10^5 s + 7066}{s^2 + 11.16s + 0.4724}$	95.6%
2	1	$\frac{1.416 \times 10^5 s + 6166}{s^2 + 11.48s + 0.4221}$	95.36%
4	3	$\frac{1.284 \times 10^5 s^3 + 2.171 \times 10^5 s^2 + 5.519 \times 10^5 s + 8.558 \times 10^4}{s^4 + 12.02s^3 + 21.85s^2 + 45.74s + 5.252}$	94.47%
4	4	$\frac{1.313 \times 10^4 s^4 + 1.82 \times 10^6 s^3 + 1.717 \times 10^8 s^2 + 1.134 \times 10^9 s + 1.164 \times 10^9}{s^4 + 153.4s^3 + 1.189 \times 10^4 s^2 + 1.191 \times 10^5 s + 6.999 \times 10^4}$	78.74%
3	0	$\frac{2.903 \times 10^8}{s^3 + 138.7s^2 + 3561s + 1.706 \times 10^4}$	70.84%
2	0	$\frac{2.448 \times 10^6}{s^2 + 27.37s + 143.7}$	70.7%

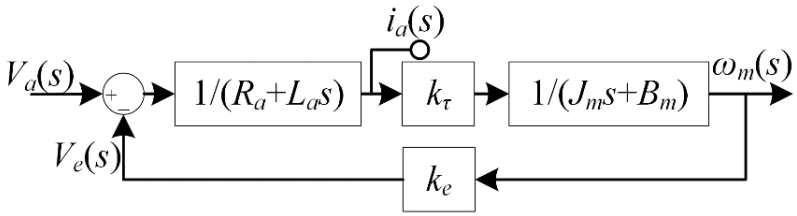


Fig. 2.3 Block diagram for the voltage to motor angular speed transfer function.

In Fig. 2.3, a block diagram for the subsystems of the motor is depicted, where it can be seen that current i_a is a virtual control that is half of the complete loop [Sahputro-17], [Alkurawy-18]. A transfer function model for the motor can be obtained using (1-12)-(1-15) as follows. First, a state space representation is obtained by defining the following states, output and input of the motor as

$$\begin{aligned} x_1(t) &= \omega_m(t) \\ x_2(t) &= i_a(t) \\ y(t) &= x_1(t) \\ u(t) &= V_a(t). \end{aligned} \tag{2-1}$$

Then, the original equations are expressed in function of the states and input, yielding

$$\dot{x}_1(t) = \frac{d\omega_m(t)}{dt} = \frac{k_\tau x_2(t) - B_m x_1(t) - \tau_l(t)}{J_m} \tag{2-2}$$

$$\dot{x}_2 = \frac{di_a(t)}{dt} = \frac{u(t) - R_a x_2(t) - k_e x_1(t)}{L_m}. \tag{2-3}$$

These equations can be transformed to a matrix form as [Ogata-96]

$$\dot{\mathbf{x}}(t) = \mathbf{A}\mathbf{x}(t) + \mathbf{B}\mathbf{u}(t), \qquad \mathbf{y}(t) = \mathbf{C}\mathbf{x}(t) + \mathbf{D}\mathbf{u}(t) \tag{2-4}$$

$$\begin{bmatrix}\dot{x}_1(t)\\ \dot{x}_2(t)\end{bmatrix} = \begin{bmatrix}-\frac{B_m}{J_m} & \frac{k_\tau}{J_m}\\ -\frac{k_e}{L_m} & -\frac{R_a}{L_m}\end{bmatrix}\begin{bmatrix}x_1(t)\\ x_2(t)\end{bmatrix} + \begin{bmatrix}0\\ \frac{1}{L_m}\end{bmatrix}u(t) + \begin{bmatrix}-\frac{1}{J_m}\\ 0\end{bmatrix}\tau_l(t) \tag{2-5}$$

$$y(t) = \begin{bmatrix}1 & 0\end{bmatrix}\begin{bmatrix}x_1(t)\\ x_2(t)\end{bmatrix}. \tag{2-6}$$

The torque $\tau_l(t)$ is assumed as a perturbation for linearity considerations, as described in 3.3.1. From this state-space representation, the associated voltage to angular speed transfer function results of the form

$$G(s) = \frac{\omega_m(s)}{V_a(s)} = \frac{\frac{k_\tau}{J_m L_m}}{s^2 + \left(\frac{B_m L_m + J_m R_a}{J_m L_m}\right)s + \frac{B_m R_a + k_e k_\tau}{J_m L_m}} \tag{2-7}$$

where all motor and vehicle parameters are defined in Table 2.3 for a quad-rotor with a QAV250 frame and EMAX[4] MT2204 2300KV motors with ESC 2-4S 12A using BLHeli firmware 14.9, 6030 propellers attached, and power supply of 11.1V 2200 mAh Lithium-Polymer battery.

TABLE 2.3. MULTI-ROTOR INPUT PARAMETERS.

Parameter	Description	Value
m	Mass of the vehicle	0.85 kg
g	Gravity force	9.80665 m/s^2
l	Arm length	0.1272 m
C_T	Thrust coefficient	81.7445×10^{-3}
C_D	Drag coefficient	45×10^{-3}
I_x	Moment of inertia in the x-axis	2.14571×10^{-3} m^2
I_y	Moment of inertia in the y-axis	3.28887×10^{-3} m^2
I_z	Moment of inertia in the z-axis	4.94533×10^{-3} m^2
k_e	Back EMF constant	6.4×10^{-4}
k_τ	Torque constant	0.432140018
R_a	Motor inertia resistance	0.05 Ω
B_m	Coefficient of viscous friction	0.000341336
J_m	Rotor inertia	1.47106×10^{-5} kg·m^2
L_m	Inductance	0.012 H

After that, using the values from the transfer function of 2 poles and 0 zeros in Table 2.2, and the parameters from [EMAX-18] and [Kumpanya-15] as a starting point, L_m and R_a were defined, allowing the calculation of J_m, B_m and k_e by means of the following equations

[4] EMAX, EMAX US Inc., 2861 Saturn St. Unit A, Brea, CA 92821, 2019.

$$\frac{k_\tau}{J_m L_m} = 2.448 \times 10^6 \tag{2-8}$$

$$\frac{B_m L_m + J_m R_a}{J_m L_m} = 27.37 \tag{2-9}$$

$$\frac{B_m R_a + k_e k_\tau}{J_m L_m} = 143.7. \tag{2-10}$$

During the test execution, it was noticed that the procedure for requesting data from the daughter-board had electrical noise due to multi-rotor frame vibration caused by the actuators. Therefore, instead of using a pin voltage level change interrupt for requesting data, the request is implemented in serial communication as defined in section 5.1.1.1.

2.2. Vehicle's Parameters Definition

The methodology used to obtain the parameters of the actuator of the vehicle is not suitable to characterize the dynamical parameters of the UAV's frame. This is due to the complexity of mounting enough sensors to measure all the variables necessary to estimate all the parameters as the vehicle mass, moments of inertia of the frame, its dimensions. Therefore, the procedure followed in this works is based on running a computer-assisted design (CAD) software and analyzing a 3D model of the QAV250 UAV frame designed in [GC-18].

On the other hand, in a lot of research works found in the literature, the thrust coefficient is normally defined without considering its physical meaning and the procedure used to obtain its value. In [Arellano-Muro-13] a thrust coefficient of 2.98×10^{-3} and a drag coefficient of 1.14×10^{-7} are presented, but there is no background explanation of the source, which is very important when it comes to motor control. The equation of thrust correlates the thrust force (T), the propeller diameter (D), the propeller/motor speed (n), the air density (ρ) and the thrust coefficient (C_T) with the following equation

$$T = C_T \rho n^2 D^4 \tag{2-11}$$

where ρ is in $\mathrm{kg/m^3}$ units, n is in $\mathrm{rev/s}$ units, D is in m units, and T is in N units, as obtained in [Jun-Li-11], [Greitzer-03], and [Felismina-17]. With regards to the drag coefficient, this correlates the drag force (F_D), the air density, the cross-section of the movement (A), the velocity (v) and the drag coefficient (C_D) with the following

$$F_D = \frac{C_D \rho A v^2}{2} \tag{2-12}$$

where F_D is in N units, A in m^2 units, and v in m/s units. Fig. 2.4 shows a diagram that exposes the force vectors generated by the propellers.

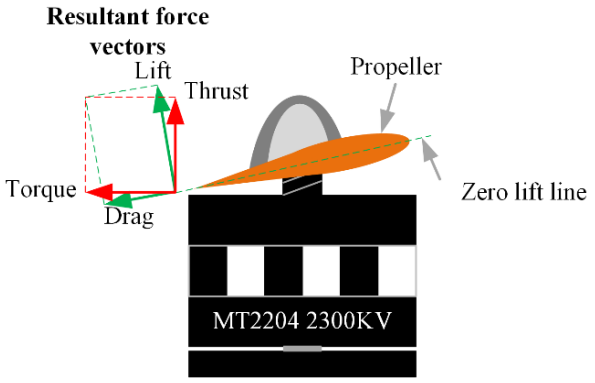


Fig. 2.4 Force vectors produced by the propellers.

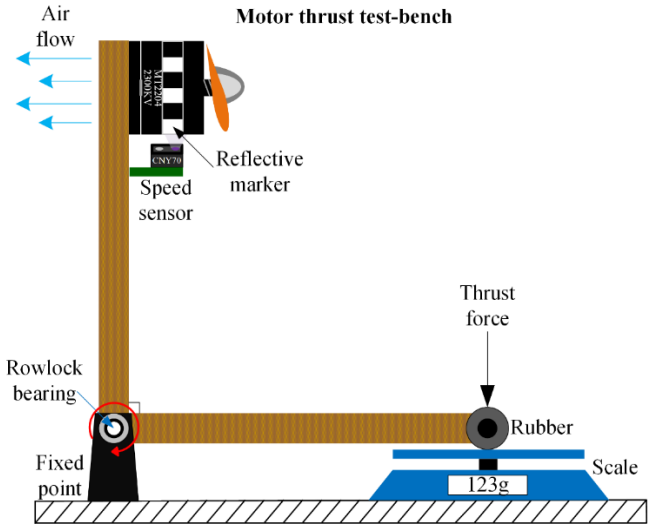


Fig. 2.5 Test-bench configuration for obtaining the thrust coefficient.

The motor manufacturer reports a thrust force value that can be obtained by using 6030 propellers with their brushless actuators, but the force measurements results show a slight difference. The use of a test-bench and a scale is required, and different speeds between the supported ranges were tested in order to create the characteristic curve for the thrust coefficient [Hafifi-Zulkipli-16], [Luque-Vega-14]. Fig. 2.5 shows the test-bench used to obtain the thrust force while increasing the motor speed and Fig. 2.6 shows the characteristic function of the measured values converted to the thrust coefficient.

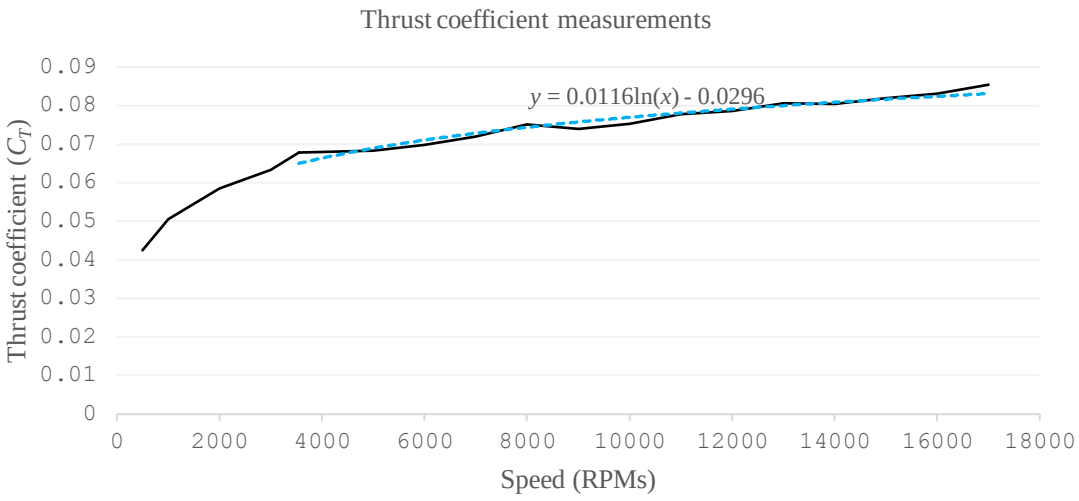


Fig. 2.6 Thrust coefficient curve based on EMAX MT2204 2300KV motor thrust force measurements.

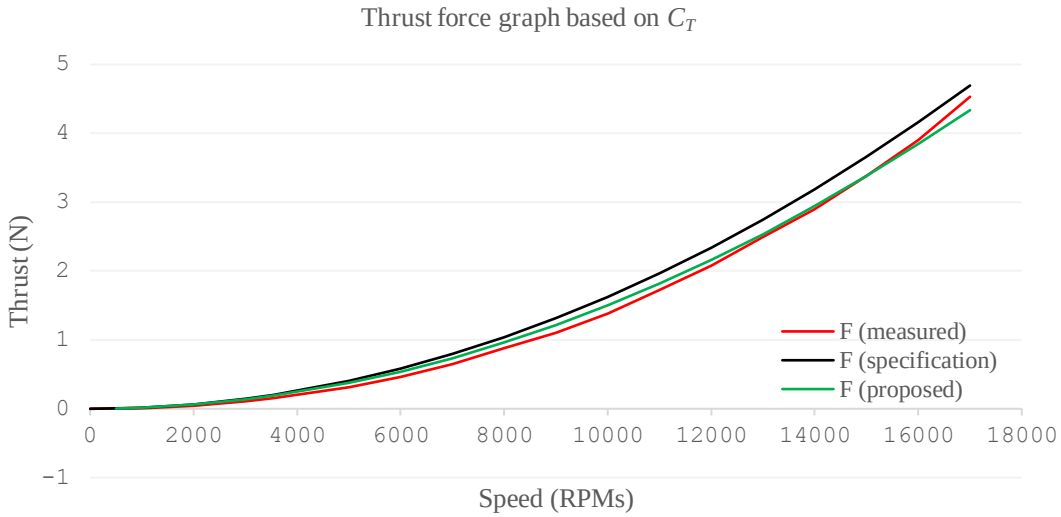
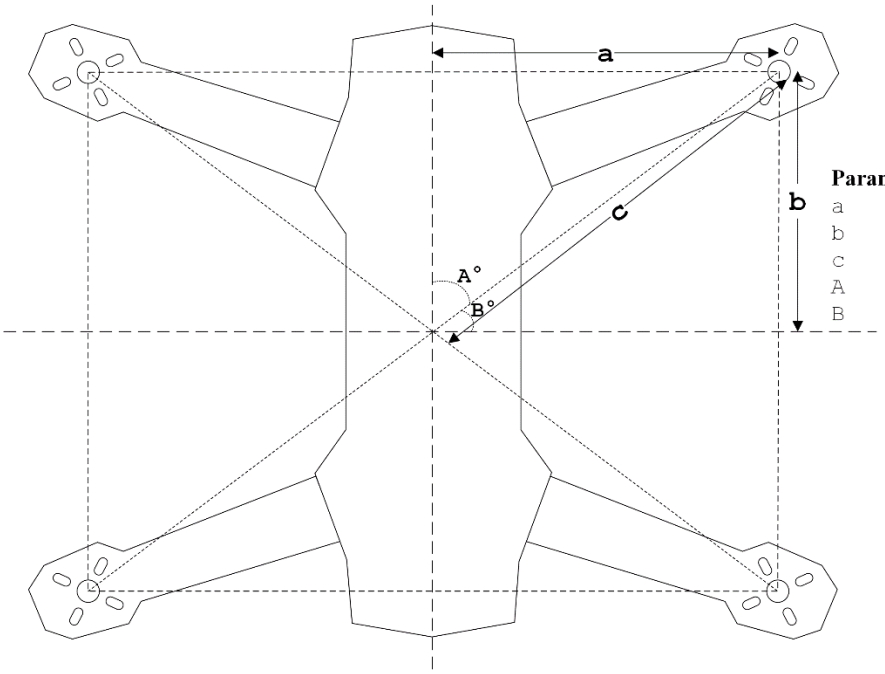


Fig. 2.7 Comparison between the thrust coefficient at different motor speeds using the measured coefficient, the motor manufacturer coefficient, and the proposed coefficient.

As it can be observed, the coefficient is a function of the speed and not a fixed value, but a function cannot be used in the control code execution as the speed reference calculation requires this to be a hardcoded value to be used by the motor controller computing. For this reason, a C_T value is proposed based on a minimum root mean square error (RMSE). Fig. 2.7 shows the thrust force obtained by using the thrust coefficient from the manufacturer values, from the measured values, and from the value proposed. With a proposed $C_T = 0.0817445$, a minimal RMSE of

0.077301668 N (0.7882576369 gf) is obtained, compared to the measured values, which is relatively small with respect to the thrust force difference.

Due to the fact that the cross-section area facing the direction of movement of the drone is a complex calculation during flight, a fixed drag coefficient is proposed, using 43×10^{-3} because it shows promising results for obtaining the controllability of the UAV.



QAV250 drone dimensions

Parameter	Value	Units
a	10.125	cm
b	7.75	cm
c	12.7506127	cm
A	52.568397	degrees
B	37.431603	degrees

Fig. 2.8 QAV250 frame dimensions.

In section 3.3 the speed reference $\omega_r(t)$ for the motor control is not indicated by the user, as the UAV reference signals, instead, $\omega_t(t)$ is a result of the control vector U, as described in (2-11) [Jun-Li-11]. The following equations obtained from Table 1.1, describe the relation between U and the reference values for the four angular velocities of the propellers

$$\begin{aligned} U_1 &= C_T\rho D^4(\omega_1^2 + \omega_2^2 + \omega_3^2 + \omega_4^2) = F_1 + F_2 + F_3 + F_4 \\ U_2 &= C_T\rho D^4(\omega_1^2 - \omega_2^2 - \omega_3^2 + \omega_4^2) = F_1 - F_2 - F_3 + F_4 \\ U_3 &= C_T\rho D^4(\omega_1^2 - \omega_2^2 + \omega_3^2 - \omega_4^2) = F_1 - F_2 + F_3 - F_4 \\ U_4 &= C_D\rho D^4(\omega_1^2 + \omega_2^2 - \omega_3^2 - \omega_4^2) = F_1 + F_2 - F_3 - F_4. \end{aligned} \tag{2-13}$$

The reason for the asymmetry observed in this configuration is due to the QAV250 frame being used, therefore, the x and y components of the forces need to be calculated to get a more accurate control execution. Fig. 2.8 shows a diagram of the dimensions of the QAV250 frame.

2.3. Simulation of the Vehicle Dynamic Model in Open-Loop

Since the control schemes developed in this research are based on the structure of the model of the controlled system, it is important to validate the dynamical model of the system to be controlled before implementing any type of controller. Aiming to demonstrate the validity and completeness of the obtained dynamical model, multiple tests in open loop are performed, analyzed, and compared to the expected behavior of the real multi-rotor.

The simulation experiment is designed as follows. First, the force necessary to counteract the weight of the vehicle is calculated as

$$F_h = \frac{mg}{n_m} = \frac{(0.85)(9.80665)}{4} = 2.083913125 \text{ N} \tag{2-14}$$

where the device mass m is obtained, that multiplied by g, provides the force that needs to be applied by the motors to hover. F_h is the force required by each motor to hover, and n_m is the number of motors in the UAV configuration. Using this result, the necessary angular velocity for the propellers of the vehicle is obtained as

$$n = \sqrt{\frac{F_h}{c_T \rho D^4}} = 196.4140 \text{ rev/s} = 11784.8431 \text{ RPM}. \tag{2-15}$$

Afterwards, a speed differential is injected in the motors in order to achieve roll, pitch, yaw, and z rotations and movements, according to the configuration diagram defined in Fig. 1.2. The velocity differential injected in each step is 30 rev/s, 30 rev/s, and 100 rev/s for roll, pitch, and yaw, respectively.

Finally, the simulation results are analyzed and compared with the expected behavior of the vehicle, considering the inputs of the experiment. Several experiments where a basic motion of the attitude and position of the vehicle were replicated. For instance, the motion generated by the propellers with a given combination of angular speeds are displayed Fig. 2.9, Fig. 2.10, and Fig. 2.11.

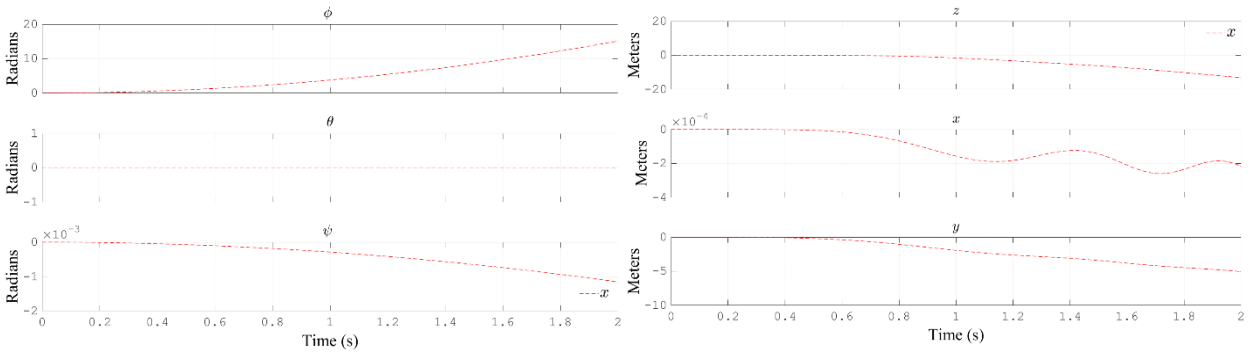


Fig. 2.9 a) Attitude motion (roll, pitch and yaw) b) translational motion (altitude, latitudinal and longitudinal) generated during a roll movement of the vehicle in open-loop.

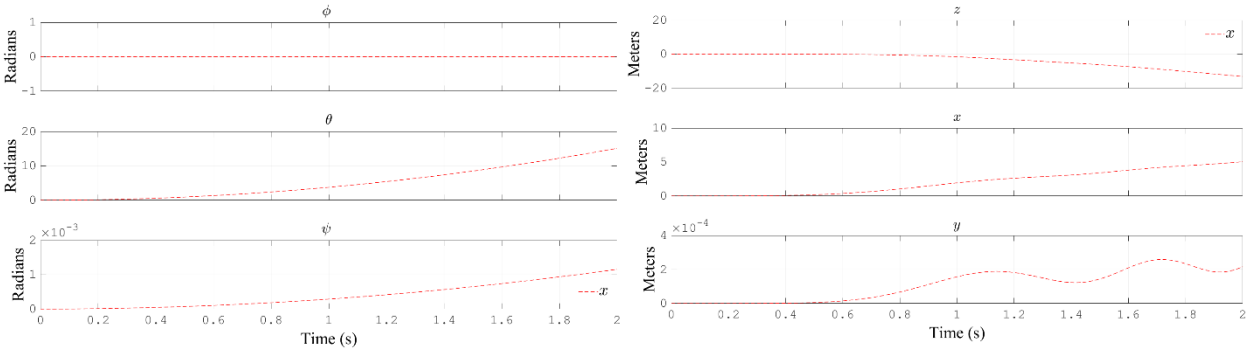


Fig. 2.10 a) Attitude motion (roll, pitch and yaw) b) translational motion (altitude, latitudinal and longitudinal) generated during a pitch movement of the vehicle in open-loop.

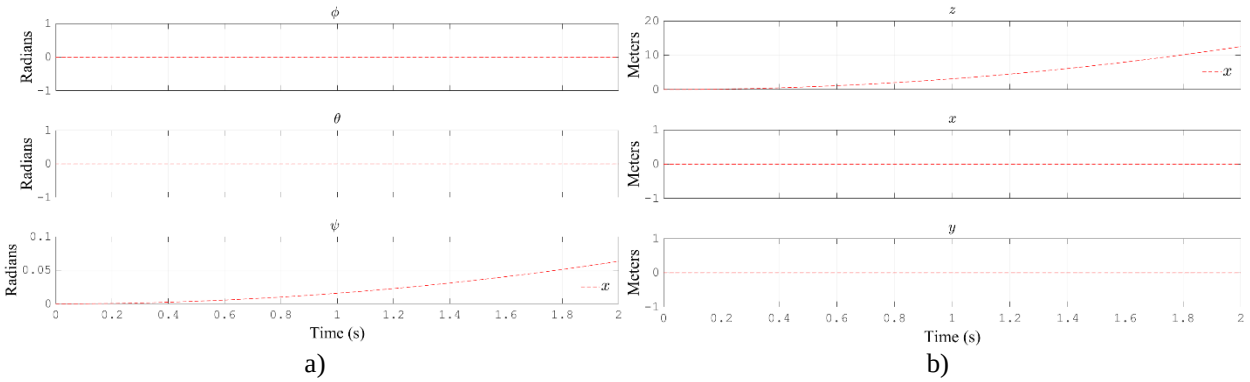


Fig. 2.11 a) Attitude motion (roll, pitch and yaw) b) translational motion (altitude, latitudinal and longitudinal) generated during a yaw movement of the vehicle in open-loop.

Fig. 2.9 shows the behavior of the vehicle when a roll movement of the vehicle is intended. It can be noted how the roll variable ϕ is increased and, as a result, the y and z coordinates of the

model are decreased. This is an expected behavior accordingly to the fixed and inertial frames defined in Fig. 1.1. Similar experiments are depicted in Fig. 2.10 and Fig. 2.11 where pitch and yaw movements of the vehicle were generated.

The complete set of open loop experiments developed permitted to validate the model defined for the vehicle and the relationships expected between its variables.

2.4. Conclusions

Characterization of the aircraft and the actuator parameters is a required step for the development of a model-based controller for a UAV. Commercial actuators and multi-rotor frame structures do not usually specify accurate dimensions, mass, body inertias, power consumption, force factors, etc. Therefore, identification of the system becomes an essential step for not introducing undesired disturbances in the embedded device, which can potentially affect the controller's output, providing a control signal that does not correspond to the vehicle being maneuvered.

To facilitate the validation of the parameters, test-benches are recommended to reduce the risk exposure for the user, and to obtain the measurements in a consistent matter for repeatability and scalability of the test for changes in the configuration. Once the parameters are known, simulation in open loop for the system is desired, in order to verify that the expected inputs produce the expected outputs, even if the control algorithms are not implemented yet.

Since UAV platforms are complex vehicles, a more sophisticated controller mechanism is required than open-loop algorithms. The next section introduces the design of the proposed controllers which are composed by an outer and an inner loop, to control the vehicle and the actuators, respectively. The proposal gives the ability to maneuver the vehicle while monitoring the status of the actuators, decreasing the chattering in the mechanical elements that drive the aircraft.

3. Design of the Proposed Controllers

In this section, the control scheme proposed for the multi-rotor is presented. In addition, the design procedure for the control algorithms is developed considering two flight modes: stabilize and trajectory tracking. The former considers references for the altitude and attitude of the UAV, and the latter considers references for the 3D position and yaw angle of the vehicle. In both cases, the control objective is to perform the reference tracking by means of the UAVs actuators. These are BLDC motors which are modelled using experimentally defined transfer functions. The control strategy considers two control loops: an inner loop for the actuators of the vehicle, and an outer loop for the UAV frame.

3.1. Overall Control Scheme

A block diagram representation of the overall control scheme is depicted in Fig. 3.1 where the two control loops of the proposal can be appreciated. The general control objective is to perform the tracking of the reference vector Y_r by the output Y of the vehicle. To this end, a UAV controller is designed which generates the necessary angular velocities of the actuators ω_{1r}, ω_{2r}, …, ω_{nr} with n as the amount of actuators of the UAV. This is defined as the outer control loop. Afterwards, the actuators controller is designed to perform the tracking of the references ω_{1r}, …, ω_{nr} by the real angular velocities of the actuators ω_r, …, ω_n which represents the inner control loop. This is achieved by defining the duty cycle of the PWM signals of the BLDC motors.

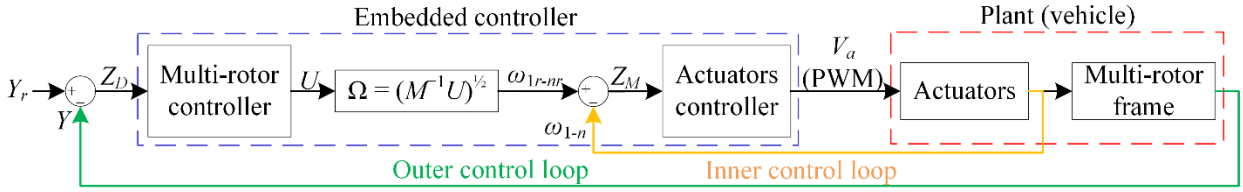


Fig. 3.1 Block diagram of the global control scheme.

The algorithms designed for the UAV controller are backstepping control, backstepping sliding mode control, and PID control [Chen-14]. On the other hand, the algorithms considered for the actuators' controller are backstepping control and PID control. The design procedure for each

of these controllers is presented in detail throughout the rest of this chapter.

3.2. Outer Control Loop

Let us recall the state space model (1-10) defined in section 1.1, expressed in the following form

$$\dot{X}\begin{bmatrix}1\\ \dots\\ 6\end{bmatrix}=\begin{bmatrix}x_2\\ f_2+b_2U_2\\ x_4\\ f_4+b_4U_3\\ x_6\\ f_6+b_6U_4\end{bmatrix},\dot{X}\begin{bmatrix}7\\ \dots\\ 12\end{bmatrix}=\begin{bmatrix}x_8\\ f_8+b_8U_1\\ x_{10}\\ u_x\left(\frac{1}{m}\right)U_1\\ x_{12}\\ u_y\left(\frac{1}{m}\right)U_1\end{bmatrix} \tag{3-1}$$

where $X=\left[\phi,\dot{\phi},\theta,\dot{\theta},\psi,\dot{\psi},z,\dot{z},x,\dot{x},y,\dot{y}\right]^T$. It can be noted that the complete model is composed of 6 subsystems with the form

$$\begin{aligned}\dot{x}_i&=x_{i+1}\\ \dot{x}_{i+1}&=f_{i+1}+b_{i+1}U_j\end{aligned} \tag{3-2}$$

with $(i,j)=\{(1,2),(3,3),(5,4),(7,1),(9,1),(11,1)\}$, which will be useful during the design stage of the controller. In the subsequent sections, these subsystems will be referenced by the i^{th} state variable related to it, i.e. subsystems 1, 3, 5, 7, 9 and 11 corresponding to the states $x_1=\phi, x_3=\theta, x_5=\psi, x_7=z, x_9=x$, and $x_{11}=y$.

As mentioned before, the UAV controller receives a reference vector Y_r for the output Y in the system. This reference vector is defined depending on the flight mode stablished for the vehicle. For instance, if a stabilize flight mode is set, the reference vector is given as $Y_r=[\phi_r,\theta_r,\psi_r,z_r]^T$. On the other hand, if a trajectory flight mode is selected, the reference vector is defined as $Y_r=[\psi_r,z_r,x_r,y_r]^T$. It is worth to note that the flight mode selection affects the UAV controller design procedure as well as the sensors considered for a real-time implementation of the control strategy.

3.2.1 Stabilize Flight Mode

In this flight mode, the reference values $x_{1r}=\phi_r$, $x_{3r}=\theta_r$, $x_{5r}=\psi_r$, and $x_{7r}=z_r$ are known and bounded, as well as their first and second derivatives. Therefore, x and y linear

movement are not directly controlled, instead, they are a consequence of the generated roll (ϕ) and pitch (θ) rotations. According to the subsystems defined in (3-2), the input terms U_{1-4} will be designed to enforce the values of $x_1 = \phi$, $x_3 = \theta$, $x_5 = \psi$, and $x_7 = z$ to their corresponding references. This is depicted in the following figure.

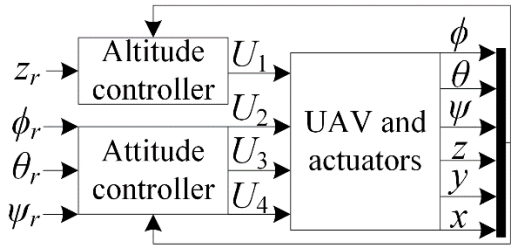


Fig. 3.2 Stabilize flight mode control scheme.

Three different control algorithms are designed to this end, and their design procedure is presented next.

3.2.1.1 Backstepping Controller

As defined in (3-2), the first 4 subsystems must be controlled by the input terms. Hence, four error variables of the system can be defined as

$$z_i = x_{ir} - x_i \tag{3-3}$$

for $i = \{1,3,5,7\}$. Then, a Lyapunov function is designed for each of the subsystem as

$$V_i = \frac{1}{2} z_i^2 \tag{3-4}$$

which is a positive definite function, as required to fulfill the stability Lyapunov's theorem [Bouabdallah-05], [Ogata].-96]. Direct differentiation of V_i results in

$$\dot{V}_i = z_i \dot{z}_i = z_i(\dot{x}_{ir} - \dot{x}_i) = z_i(\dot{x}_{ir} - x_{i+1}) \tag{3-5}$$

that must be a negative semi-definite function to assure the stability of the closed loop system. In order to do so, the term x_{i+1}^* is used as a virtual control term and it is proposed as

$$x_{i+1}^* = \dot{x}_{ir} + k_i z_i. \tag{3-6}$$

Now, this virtual control x_{i+1}^* is used as a reference for x_{i+1} which allows us to define the error variable

$$z_{i+1} = x_{i+1}^* - x_{i+1} \tag{3-7}$$

and, by means of (3-2) and (3-6), its derivative yields

$$\dot{z}_{i+1} = \ddot{x}_{ir} + k_i \dot{z}_i - f_{i+1} - b_{i+1} U_j \tag{3-8}$$

with the ordered pairs $(i, j) = \{(1,2), (3,3), (5,4), (7,1)\}$.

Again, a Lyapunov's function is defined for z_{i+1} as

$$V_{i+1} = V_i + \frac{1}{2} z_{i+1}^2 \tag{3-9}$$

whose derivative is obtained of the form

$$\dot{V}_{i+1} = \dot{V}_i + z_{i+1} \dot{z}_{i+1}. \tag{3-10}$$

Using (3-5)-(3-8), the previous equation transforms to

$$\dot{V}_{i+1} = z_i (z_{i+1} - k_i z_i) + z_{i+1} \left(\ddot{x}_{ir} + k_i \dot{z}_i - f_{i+1} - b_{i+1} U_j \right). \tag{3-11}$$

Finally, the control signal U_j is designed as

$$U_j = b_{i+1}^{-1} (\ddot{x}_{ir} + k_i \dot{z}_i - f_{i+1} + z_i + k_{i+1} z_{i+1}) \tag{3-12}$$

which finalizes the control design procedure.

One of the advantages of the backstepping controller, is that once the control laws design is completed, the stability analysis of the closed loop system is straightforward. In this case, by substituting the designed control laws (3-6) and (3-12), the terms $\dot{V}_i$ and $\dot{V}_{i+1}$ are simplified to

$$\begin{aligned} \dot{V}_i &= -k_i z_i^2 + z_i z_{i+1} \\ \dot{V}_{i+1} &= -k_i z_i^2 - k_{i+1} z_{i+1}^2. \end{aligned} \tag{3-13}$$

Then, the linear dynamic system (3-11) can be expressed as

$$\begin{bmatrix} \dot{V}_i \\ \dot{V}_{i+1} \end{bmatrix} = - \begin{bmatrix} 0 \\ k_i z_i^2 \end{bmatrix} - \begin{bmatrix} z_i & z_{i+1} \end{bmatrix} \begin{bmatrix} k_i & 1 \\ 0 & k_{i+1} \end{bmatrix} \begin{bmatrix} z_i \\ z_{i+1} \end{bmatrix} \tag{3-14}$$

and, assuming $k_i > 0$ and $k_{i+1} > 0$, the inequalities $\dot{V}_i < 0$ and $\dot{V}_{i+1} < 0$ are demonstrated for $z_i \neq 0$ and $z_{i+1} \neq 0$. Consequently, the Lyapunov's theorem of stability is fulfilled, the error variables z_i and z_{i+1} converge to zero asymptotically, and the control objective is achieved.

Finally, the control vector U can be expressed in terms of the states, error variables and the systems' parameters as

$$U = \begin{bmatrix} U_1 \\ U_2 \\ U_3 \\ U_4 \end{bmatrix} = \begin{bmatrix} \dfrac{m}{\cos x_1 \cos x_3} (\ddot{x}_{7r} + k_7 \dot{z}_7 + g + z_7 + k_8 z_8) \\ \dfrac{1}{b_1} (\ddot{x}_{1r} + k_1 \dot{z}_1 - x_4 x_6 a_1 - x_4 a_2 \Omega + z_1 + k_2 z_2) \\ \dfrac{1}{b_2} (\ddot{x}_{3r} + k_3 \dot{z}_3 - x_2 x_6 a_3 - x_2 a_4 \Omega + z_3 + k_4 z_4) \\ \dfrac{1}{b_3} (\ddot{x}_{5r} + k_5 \dot{z}_5 - x_4 x_2 a_5 + z_5 + k_6 z_6) \end{bmatrix}. \tag{3-15}$$

3.2.1.2 Backstepping Sliding Mode Controller

For this controller, the same procedure used for the backstepping controller is followed. The only difference is that the term $k_{i+1}z_{i+1}$ in the control inputs U_j is replaced by the discontinuous term $k_{i+1}\text{sign}(z_{i+1})$ which is characteristic of the sliding mode control technique. Henceforth, the control signal U_j is designed as

$$U_j = b_{i+1}^{-1}\left(\ddot{x}_{ir} + k_i\dot{z}_i - f_{i+1} + z_i|z_{i+1}| + k_{i+1}\text{sign}(z_{i+1})\right) \tag{3-16}$$

and the control design procedure is finished.

Regarding the stability analysis, the Lyapunov functions candidates are defined as

$$V_i = \frac{1}{2}z_i^2 \text{ and } V_{i+1} = V_i + |z_{i+1}|. \tag{3-17}$$

Hence, considering (3-16), their derivatives are obtained as

$$\begin{aligned} \dot{V}_i &= -k_i z_i^2 + z_i z_{i+1} \\ \dot{V}_{i+1} &= -k_i z_i^2 - k_{i+1} \end{aligned} \tag{3-18}$$

which, again assuming $k_i > 0$ and $k_{i+1} > 0$, the inequalities $\dot{V}_i < 0$ and $\dot{V}_{i+1} < 0$ are demonstrated for $z_i \neq 0$ and $z_{i+1} \neq 0$. Consequently, the Lyapunov's theorem of stability is fulfilled, the error variables z_i and z_{i+1} converge to zero asymptotically, and the control objective is achieved.

Finally, the control vector U can be expressed in terms of the states, error variables and the systems' parameters as

$$U = \begin{bmatrix} U_1 \\ U_2 \\ U_3 \\ U_4 \end{bmatrix} = \begin{bmatrix} \dfrac{m}{\cos x_1 \cos x_3}\left(\ddot{x}_{7r} + k_7\dot{z}_7 + g + z_7|z_8| + k_8\text{sign}(z_8)\right) \\ \dfrac{1}{b_1}\left(\ddot{x}_{1r} + k_1\dot{z}_1 - x_4x_6a_1 - x_4a_2\Omega + z_1|z_2| + k_2\text{sign}(z_2)\right) \\ \dfrac{1}{b_2}\left(\ddot{x}_{3r} + k_3\dot{z}_3 - x_2x_6a_3 - x_2a_4\Omega + z_3|z_4| + k_4\text{sign}(z_4)\right) \\ \dfrac{1}{b_3}\left(\ddot{x}_{5r} + k_5\dot{z}_5 - x_2x_4a_5 + z_5|z_6| + k_6\text{sign}(z_6)\right) \end{bmatrix}. \tag{3-19}$$

3.2.1.3 PID Controller

This controller is based on the derivative, integral and proportional effect of error feedback. In the same manner of the two previous controllers, one control term of U_{1-4} is assigned for the reference tracking of one of the outputs defined for stabilize flight mode $x_1 = \phi, x_3 = \theta, x_5 = \psi$,

and $x_7 = z$. Hence, considering the references $Y_r = [x_{1r}, x_{3r}, x_{5r}, x_{7r}]$ for this flight mode, the following error variables can be defined

$$e_\phi = x_{1r} - x_1, \quad e_\theta = x_{3r} - x_3, \quad e_\psi = x_{5r} - x_5, \quad e_z = x_{7r} - x_7. \tag{3-20}$$

Then, the control vector U can be expressed in terms of the error variables as

$$U = \begin{bmatrix} U_1 \\ U_2 \\ U_3 \\ U_4 \end{bmatrix} = \begin{bmatrix} K_{P1} e_z + K_{I1} \int_0^t e_z \, dt + K_{D1} \dot{e}_z \\ K_{P2} e_\phi + K_{I2} \int_0^t e_\phi \, dt + K_{D2} \dot{e}_\phi \\ K_{P3} e_\theta + K_{I3} \int_0^t e_\theta \, dt + K_{D3} \dot{e}_\theta \\ K_{P4} e_\psi + K_{I4} \int_0^t e_\psi \, dt + K_{D4} \dot{e}_\psi \end{bmatrix} \tag{3-21}$$

where K_{Pi}, K_{Ii}, K_{Di} for $i = \{1,2,3,4\}$ are the controller gains to be defined. This tuning process was performed manually for this work and the results will be presented in the implementation section of the document.

3.2.2 Trajectory Flight Mode

In this flight mode, the reference values $x_{5r} = \psi_r$, $x_{7r} = z_r$, $x_{9r} = x_r$ and $x_{11r} = y_r$ are known and bounded, as well as their first and second derivatives. Therefore, x and y linear movements are indirectly controlled to obtain a desired position for the UAV. Similar to the stabilize flight mode, the subsystems 5 and 7 defined in (3-2) will be controlled by U_4 and U_1. Moreover, the controller design procedure for U_4 and U_1 will remain the same for this flight mode. Regarding the subsystems 9 and 11, the control inputs U_2 and U_3 cannot be used to directly control $x_9 = x$ and $x_{11} = y$ as the control terms do not appear explicitly in the dynamical equations of these subsystems. This strategy is depicted in the following figure [Fan-17].

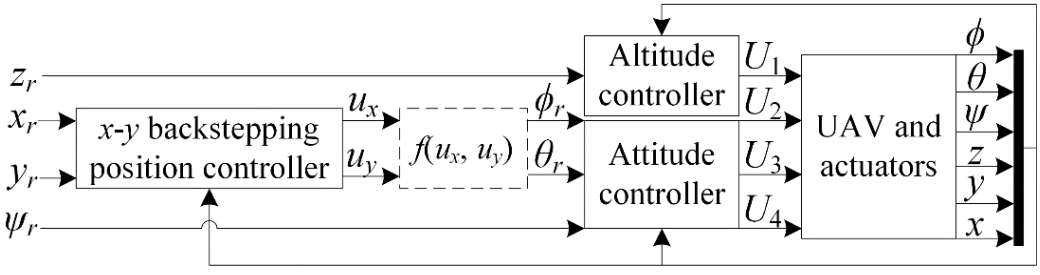


Fig. 3.3 Trajectory flight mode control scheme.

Therefore, the control strategy is proposed according to the following steps:

S1. The terms u_x, u_y are proposed as virtual control terms for the subsystems 9 and 11

S2. Then, the obtained u_x, u_y are used to define the references for the states $x_1 = \phi$ and $x_3 = \theta$, and these references x_{1r} and x_{3r} are used to define the error variables $z_1 = x_{1r} - x_1$ and $z_3 = x_{3r} - x_3$.

S3. The control terms U_2 and U_3 are designed to control the subsystems 1 and 3 in the same manner that the stabilize flight mode.

The step S2 is resolved by using equation (1-10). From there, the following equations can be obtained to relate the references for x_1 and x_3 with the terms u_x and u_y

$$x_{1r} = \sin^{-1}\left(u_x \sin x_5 - u_y \cos x_5\right), \quad x_{3r} = \sin^{-1}\left(\frac{u_x - \sin x_1 \sin x_5}{\cos x_1 \cos x_5}\right). \tag{3-22}$$

The step S3 was already developed for the stabilize flight mode. The resulting control terms were defined in (3-15), (3-19) and (3-21) and will be used for trajectory flight mode as well.

Hence, the remaining of this subsection is dedicated to address the development of the step S1 for the three proposed controllers.

3.2.2.1 Backstepping Controller

As previously stablished, the control design procedure needs to address only the subsystems 9 and 11. Hence, it is assumed that the references x_{9r} and x_{11r} are known a priori and their related error variables can be defined of the form

$$z_i = x_{ir} - x_i \tag{3-23}$$

for $i = \{9,11\}$. Then, a Lyapunov function is designed for each of the subsystem as

$$V_i = \frac{1}{2} z_i^2 \tag{3-24}$$

which is a positive definite function, as required to fulfill the stability Lyapunov's theorem [Ogata-96]. Direct differentiation of V_i results in

$$\dot{V}_i = z_i \dot{z}_i = z_i(\dot{x}_{ir} - \dot{x}_i) = z_i(\dot{x}_{ir} - x_{i+1}) \tag{3-25}$$

that must be a negative semi-definite function to assure the stability of the closed loop system. In order to do so, the term x_{i+1} is used as a virtual control term and it is proposed as

$$x_{i+1}^* = \dot{x}_{ir} + k_i z_i. \tag{3-26}$$

Now, this virtual control x_{i+1}^* is used as a reference for x_{i+1} which allows us to define the error variable

$$z_{i+1} = x_{i+1}^{*} - x_{i+1} \tag{3-27}$$

and, by means of (3-2) and (3-6), its derivative yields

$$\dot{z}_{i+1} = \ddot{x}_{ir} + k_i \dot{z}_i - b_0 U_{i+1} \tag{3-28}$$

where $b_0 = \frac{U_1}{m}$, $U_{10} = u_x$ and $U_{12} = u_y$. After that, a Lyapunov function candidate is defined for z_{i+1} as

$$V_{i+1} = V_i + \frac{1}{2} z_{i+1}^2 \tag{3-29}$$

whose derivative is obtained of the form

$$\dot{V}_{i+1} = \dot{V}_i + z_{i+1} \dot{z}_{i+1}. \tag{3-30}$$

Using (3-25)-(3-28), the previous equation transforms to

$$\dot{V}_{i+1} = z_i (z_{i+1} - k_i z_i) + z_{i+1} (\ddot{x}_{ir} + k_i \dot{z}_i - b_0 U_{i+1}). \tag{3-31}$$

Finally, the control signal U_{i+1} is designed as

$$U_{i+1} = b_0^{-1} (\ddot{x}_{ir} + k_i \dot{z}_i + z_i + k_{i+1} z_{i+1}) \tag{3-32}$$

or, equivalently, as

$$u_x = b_0^{-1} (\ddot{x}_{9r} + k_9 \dot{z}_9 + z_9 + k_{10} z_{10}) \tag{3-33}$$

$$u_y = b_0^{-1} (\ddot{x}_{11r} + k_{11} \dot{z}_{11} + z_{11} + k_{12} z_{12}) \tag{3-34}$$

which finalizes the backstepping control part in the design procedure.

It is easy to see that the stability analysis for these closed-loop subsystems corresponds exactly to the one developed for the stabilize flight mode. Therefore, it can be concluded that the stability conditions for the control laws (3-32) are $k_i > 0$ and $k_{i+1} > 0$ for $i = \{9,11\}$.

Finally, the control terms U_1, U_2, U_3 and U_4 are defined exactly as in (3-15)

$$U = \begin{bmatrix} U_1 \\ U_2 \\ U_3 \\ U_4 \end{bmatrix} = \begin{bmatrix} \frac{m}{\cos x_1 \cos x_3} (\ddot{x}_{7r} + k_7 \dot{z}_7 + g + z_7 + k_8 z_8) \\ \frac{1}{b_1} (\ddot{x}_{1r} + k_1 \dot{z}_1 - x_4 x_6 a_1 - x_4 a_2 \Omega + z_1 + k_2 z_2) \\ \frac{1}{b_2} (\ddot{x}_{3r} + k_3 \dot{z}_3 - x_2 x_6 a_3 - x_2 a_4 \Omega + z_3 + k_4 z_4) \\ \frac{1}{b_3} (\ddot{x}_{5r} + k_5 \dot{z}_5 - x_4 x_2 a_5 + z_5 + k_6 z_6) \end{bmatrix} \tag{3-35}$$

along with (3-22) and (3-32).

3.2.2.2 Backstepping Sliding Mode Controller

This controller is a variation of the previous one. The only difference is that the control

laws are composed of (3-22) and (3-32), but with the definition of U_1, U_2, U_3 and U_4 of the form

$$U = \begin{bmatrix} U_1 \\ U_2 \\ U_3 \\ U_4 \end{bmatrix} = \begin{bmatrix} \frac{m}{\cos x_1 \cos x_3}\left(\ddot{x}_{7r} + k_7\dot{z}_7 + g + z_7|z_8| + k_8\text{sign}(z_8)\right) \\ \frac{1}{b_1}\left(\ddot{x}_{1r} + k_1\dot{z}_1 - x_4x_6a_1 - x_4a_2\Omega + z_1|z_2| + k_2\text{sign}(z_2)\right) \\ \frac{1}{b_2}\left(\ddot{x}_{3r} + k_3\dot{z}_3 - x_2x_6a_3 - x_2a_4\Omega + z_3|z_4| + k_4\text{sign}(z_4)\right) \\ \frac{1}{b_3}\left(\ddot{x}_{5r} + k_5\dot{z}_5 - x_4x_2a_5 + z_5|z_6| + k_6\text{sign}(z_6)\right) \end{bmatrix}. \quad (3\text{-}36)$$

3.2.2.3 Backstepping PID Controller

In section 3.2.2 it is explained that x and y linear movements cannot be controlled directly, instead, a virtual control design is required aiming to produce references ϕ_r and θ_r that can now be directly controlled to achieve trajectory tracking flight mode. The overall resulting control scheme is depicted in Fig. 3.3. Hence, in this section the backstepping technique for the virtual control of x and y is proposed, first addressing only the subsystems 9 and 11 of (3-2) where the references x_{9r} and x_{11r} are known a priori. Using the backstepping technique for these subsystems, the virtual controllers result in the following equations

$$u_x = b_0^{-1}(\ddot{x}_{9r} + k_9\dot{z}_9 + z_9 + k_{10}z_{10}) \quad (3\text{-}37)$$

$$u_y = b_0^{-1}(\ddot{x}_{11r} + k_{11}\dot{z}_{11} + z_{11} + k_{12}z_{12}). \quad (3\text{-}38)$$

Once the desired values for u_x and u_y are computed, the reference terms $x_{1r} = \phi_r$ and $x_{3r} = \theta_r$ are calculated by means of equation (1-10(1-10) as

$$x_{1r} = \sin\left(\frac{u_x\tan(x_5) - u_y}{\sin(x_5)\tan(x_5) + \cos(x_5)}\right)^{-1} \quad (3\text{-}39)$$

$$x_{3r} = \sin\left(\frac{(u_x - \sin(x_1))\sin(x_5)}{\cos(x_1)\cos(x_5)}\right)^{-1}. \quad (3\text{-}40)$$

As the trajectory tracking flight mode requires to know the references $x_{5r} = \psi_r$, and $x_7 = z_r$ beforehand, the complete reference vector $Y_r = [x_{1r}, x_{3r}, x_{5r}, x_{7r}]$ is defined. So, the associated error variables result of the form

$$e_\phi = x_{1r} - x_1, \quad e_\theta = x_{3r} - x_3, \quad e_\psi = x_{5r} - x_5, \quad e_z = x_{7r} - x_7 \quad (3\text{-}41)$$

and the control vector U can be expressed in terms of the error variables as

$$U = \begin{bmatrix} U_1 \\ U_2 \\ U_3 \\ U_4 \end{bmatrix} = \begin{bmatrix} K_{P1}e_z + K_{I1}\int_0^t e_z\,dt + K_{D1}\dot{e}_z \\ K_{P2}e_\phi + K_{I2}\int_0^t e_\phi\,dt + K_{D2}\dot{e}_\phi \\ K_{P3}e_\theta + K_{I3}\int_0^t e_\theta\,dt + K_{D3}\dot{e}_\theta \\ K_{P4}e_\psi + K_{I4}\int_0^t e_\psi\,dt + K_{D4}\dot{e}_\psi \end{bmatrix} \quad (3\text{-}42)$$

where K_{Pi}, K_{Ii}, K_{Di} for $i = \{1,2,3,4\}$ are the controller gains to be tuned. This tuning process was performed manually for this work and the results will be presented in the implementation section of the document.

3.3. Inner Control Loop

The outputs of the outer loop controller are the control terms U_1, U_2, U_3 and U_4, which can be used along with (1-6) to obtain the necessary angular velocities of the actuators as follows

$$\begin{bmatrix} \omega_1^2(t) \\ \omega_2^2(t) \\ \dots \\ \omega_n^2(t) \end{bmatrix} = M^{-1}\begin{bmatrix} U_1 \\ U_2 \\ U_3 \\ U_4 \end{bmatrix}. \quad (3\text{-}43)$$

where matrix M is defined as

$$M = \begin{bmatrix} b & b & b & b \\ p & -p & -p & p \\ q & -q & q & -q \\ d & d & -d & -d \end{bmatrix}, \begin{matrix} b = C_T\rho D^4 \\ p = b\sin(52.56°) \\ q = b\sin(37.44°) \\ d = C_D\rho A/2 \end{matrix} \quad (3\text{-}44)$$

according to (2-13) and the configuration of the UAV with frame QAV250, which will be used for the real time experiments. It is worth to note that M is a full rank matrix, assuring that there is a unique solution of the equation system (3-43). Also, the angular velocities of the propellers are strictly positive as the drivers of the actuators only generate torque in one sense of rotation.

After that, the resulting $\omega_1(t)$, $\omega_2(t)$, ..., $\omega_n(t)$ are regarded as the references $\omega_{1r}(t)$, $\omega_{2r}(t)$, ..., $\omega_{nr}(t)$ for the inner loop controller, as these variables cannot be directly manipulated.

3.3.1 Backstepping Controller

Once the velocity references $\omega_{1r}(t)$, $\omega_{2r}(t)$, ..., $\omega_{nr}(t)$ for the BLDC motors are defined, the next step is to design the inner control loop whose objective is to manipulate the voltage applied to each of the motors to generate their respective desired angular velocities. As the actuators are defined by the same exact model, only one controller must be designed and replicated for the four actuators.

First, let us recall from section 1.2 that the dynamical equations of the electric motor are given by

$$V_i(t) = R_i i_{ai}(t) + L_{mi}\frac{di_{ai}(t)}{dt} + k_{ei}\omega_{mi}(t) \tag{3-45}$$

$$k_{\tau i} i_{ai}(t) = J_{mi}\frac{d\omega_i(t)}{dt} + B_{mi}\omega_i(t) + \tau_{li}(t). \tag{3-46}$$

As the value of the inductance L_m is considerably small (in the order of mH units) with respect to the other motor parameters, the inductance is neglected and equation (3-45) is converted to

$$i_{ai}(t) = \frac{V_i(t) - k_{ei}\omega_i(t)}{R_i}. \tag{3-47}$$

Moreover, the load torque $\tau_{li}(t)$ is considered as a perturbation in the equation and set to 0. Using (3-46) and (3-47), the resulting dynamics are of the form

$$\dot{\omega}_i(t) = \frac{k_{\tau i}V_i(t)}{J_{mi}R_i} - \frac{k_{\tau i}k_{ei}\omega_i(t)}{J_{mi}R_i} - \frac{B_{mi}\omega_i(t)}{J_{mi}}. \tag{3-48}$$

Considering that the reference for the angular velocity of the i^{th} actuator is ω_{ir} for $i = \{1,2,3,4\}$, then the error variable for that actuator can be defined as

$$z_{iM} = \omega_{ir}(t) - \omega_i(t) \tag{3-49}$$

where $\omega_i(t)$ is the current angular velocity of the motor. Then, the error dynamics can be obtained of the form

$$\dot{z}_{iM} = \dot{\omega}_{ir}(t) - \dot{\omega}_i(t) = \dot{\omega}_{ir}(t) - \frac{k_{\tau i}V_i(t)}{J_{mi}R_i} + \frac{k_{\tau i}k_{ei}\omega_i(t)}{J_{mi}R_i} + \frac{B_{mi}\omega_i(t)}{J_{mi}}. \tag{3-50}$$

With the error defined, the following candidate Lyapunov function is proposed

$$V_{iM} = \frac{1}{2}z_{iM}^2. \tag{3-51}$$

This is a positive definite function, that satisfies the stability Lyapunov's theorem [Bouabdallah-05], [Ogata-96] and its derivative, obtained by direct differentiation, results in

$$\dot{V}_{iM} = z_{iM}\dot{z}_{iM} = z_{iM}\left(\dot{\omega}_{ir}(t) - \frac{k_{\tau i}V_i(t)}{J_{mi}R_i} + \frac{k_{\tau i}k_{ei}\omega_i(t)}{J_{mi}R_i} + \frac{B_{mi}\omega_i(t)}{J_{mi}}\right). \tag{3-52}$$

Now, the voltage control term $V_a(t)$ is designed of the form

$$V_i(t) = \frac{J_{mi}R_i}{k_{\tau i}}\left(\dot{\omega}_{ir}(t) + \frac{k_{\tau i}k_{ei}\omega_i(t)}{J_{mi}R_i} + \frac{B_{mi}\omega_i(t)}{J_{mi}} + k_{iM}z_{iM}\right). \tag{3-53}$$

After that, using (3-52) and (3-53), the term $\dot{V}_{iM}$ yields

$$\dot{V}_{iM} = -k_{iM}z_{iM}^2 \tag{3-54}$$

and, considering $k_{iM} > 0$, the condition $\dot{V}_{iM} < 0$ is fulfilled. Hence, by means of the Lyapunov's stability theorem, the asymptotic convergence of z_{iM} to zero is assured. Consequently, $\omega_i(t)$ converges to $\omega_{ir}(t)$ and the control objective is fulfilled.

3.3.2 Levant's Differentiator

Once the inner control loop has been designed, a new issue arises as the derivative of the reference terms $\dot{\omega}_{ir}(t)$ must be computed to fully define the control laws. The first attempt was to use the differentiator block of Simulink[5], but its performance was poor as it injected noise to the algorithm and led the simulation to fail. In order to solve these problems, a robust exact differentiator proposed by Levant [Levant-98], [Levant-03] was used instead. The main problem with classical differentiator designs is the combination of exactness and robustness with respect to possible measurement errors and input noise. Moreover, if the input signal present disturbances, the differentiator's output is contaminated too, and there is a risk of amplifying the error. In contrast to Simulink's differentiator, the Levant's differentiator provides the square of the maxima of the measured input signal from the base signal, reducing the rate error that can be present in the output [Milosavljević-10]. For the proposed controller, a second-order Levant's differentiator is used, which is defined by

$$\dot{x} = v_{n-1} \tag{3-55}$$

$$\dot{z}_0 = -\lambda_2|z_0 - f(t)|^{\frac{2}{3}}\text{sign}\big(z_0 - f(t)\big) + z_1 \tag{3-56}$$

$$\dot{z}_1 = -\lambda_1|z_1 - v_0|^{\frac{1}{2}}\text{sign}(z_1 - v_0) + z_2 \tag{3-57}$$

[5] MATLAB, Version 2017b, The MathWorks, Inc., 3 Apple Hill Drive, Natick, MA 01760-2098, 2017.

$$\dot{z}_2 = -\lambda_0 \text{sign}(z_2 - v_1) \tag{3-58}$$

where $f(t)$ is the input signal, and z_n is the estimation for the n-order derivate of $f(t)$, as shown in Fig. 3.4.

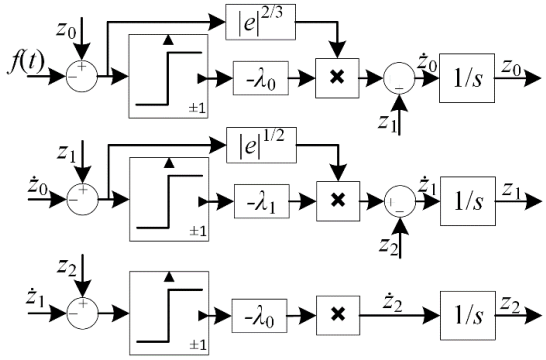


Fig. 3.4 Second-order Levant's exact differentiator block diagram.

3.4. Conclusions

For the overall controller design and implementation, a two-loop control scheme as selected. It is composed of an outer control loop for the vehicle dynamics, and an inner control loop for the actuators of the UAV. One advantage of this proposal is that the high-frequency components of the control term generated by the outer loop are filtered by the dynamics of the actuator in the simulation stage. Hence, the inner control loop, which requires the model of the BLDC motors, defines the response of the actuators in accordance with their mechanical capabilities.

Three controller designs are developed, being the backstepping controller the one chosen for real time implementation. This decision is made based on the fact that the non-linearity of the sign function from the sliding mode injects non linearities to the subsequent derivatives required for the inner loop, causing indetermination in the calculation of the controller, that result in instabilities in the execution [Derafa-12]; furthermore, the high frequency responses observed in this type of controllers is not a desired input for the brushless motors. Regarding the PID controller, avoiding the fact that the embedded implementation just requires de error calculation and its derivative and integrals for each variable to control, the lack of robustness makes it a not desired solution due to the unmodeled dynamics and the disturbances that the control method will not be able to handle.

To verify the performance of the developed controllers, the following section describes the simulation results of the system in closed loop. This is a necessary stage, aiming to validate the controller and its parameters. For that purpose, the simulation environment is setup to test the system without the risk of damaging the device, to provide a first approach of the behavior, and to tune the controller gains.

4. Closed Loop Simulation Results

Before implementing the controller in an embedded hardware, it is important to obtain simulation results in order to identify issues with the code or the model that can be fixed early in the implementation to avoid debug efforts. In that regard, the controllers designed, the dynamic models for the vehicle, and its actuators obtained in previous sections are implemented in a simulation environment. For that purpose, the Matlab/Simulink software is used as it provides pre-defined blocks for performing different calculations or actions, which do normally require time-consuming tasks when implementing them in hardware.

It is important to notice that the controller-motor interaction in the system corresponds to a stiff system behavior, which is basically a plant/actuator with slow dynamics compared with the fast processing of the controller of the plant. In other words, the time constant of the dynamic mechanical system is higher (slower) than the time constant for the controller response. The key to resolve the synchronization problem is to use advanced signal processing techniques, and algorithms to take advantage of the computational power in order to achieve the control goal no matter of the mechanical restrictions [Ehsani-99].

4.1. Block Definitions for Vehicle Dynamics Simulation

The integrated development environment (IDE) selected to simulate the closed loop response of the designed controllers, modelled UAV, and actuators, is Simulink. A general view of the blocks implemented in this IDE are shown in Fig. 4.1 where the essential subsystems of the model are indicated. For instance, the ***UAV Input Parameters*** subsystem defines all the parameters of the vehicle and its configuration. These parameters are used by the ***UAV Mathematical Model*** subsystem which contains the definition of the dynamical models of the vehicle and the actuators. It implements the system defined in (1-11) and considers the output of ***UAV control*** and the speed output from ***Motor control*** to estimate the states. Fig. C.4 shows the code related to this routine. Finally, the ***Control Algorithm*** subsystem defines the control algorithms, for the inner and outer loops, designed in the previous section.

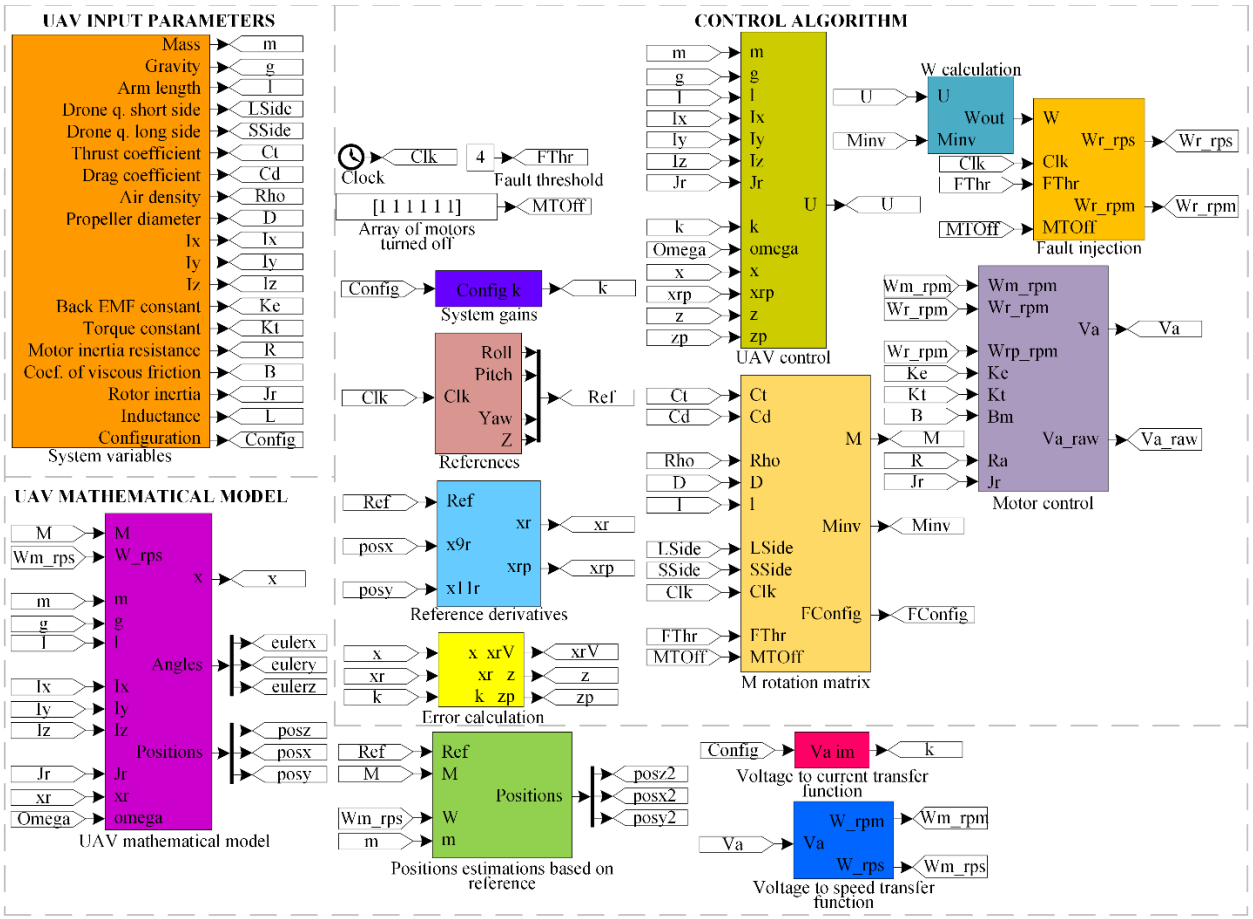


Fig. 4.1 Simulink closed loop model for the stabilize flight mode of the UAV.

For the simulation, the reference signals for the vehicle are the attitude (roll ϕ, pitch θ, and yaw ψ) and altitude (z) as the stabilize flight mode is considered. These references generate a rectangular route for the position of the vehicle and are defined as

$$\phi_r = \begin{cases} c_1 \sin t \; \forall t \in A \\ -c_1 \sin t \; \forall t \in B, \\ 0 \forall t \in C \end{cases} \tag{4-1}$$

$$\theta_r = \begin{cases} c_1 \sin t \; \forall t \in D \\ -c_1 \sin t \; \forall t \in E, \\ 0 \forall t \in F \end{cases} \tag{4-2}$$

$$\psi_r = 0, \tag{4-3}$$

$$z_r = \frac{c_1}{\left(1 + e^{-c_2\left(t - \frac{2}{c_2}\right)}\right)^2} \tag{4-4}$$

where c_1 and c_2 are constants, $A = [\pi, 2\pi)$, $B = [5\pi, 7\pi)$, $C = U - (A + B)$, $D = [7\pi, \pi)$, $E = [3\pi, 4\pi)$, and $F = U - (D + E)$. It is worth to address that the reference signals were designed to be composed by time derivable functions to avoid singularities as the control laws calculation

requires the derivatives of the references for the outputs of the system.

4.2. Outer Loop Controllers

Using the same Simulink model, the three controllers for the vehicle are simulated in closed loop. The stabilize flight mode and the same references for the attitude and altitude of the vehicle, which were defined in the previous section, are used for the three control algorithms. The results of these experiments are presented in the following section of the document.

4.2.1 Backstepping Controller

The controller gains used for this simulation experiment are given by

$$k_i = 5, k_{i+1} = 20, k_7 = 5, k_8 = 3, k_{jM} = 4000 \tag{4-5}$$

for $i = \{1,2,3\}$ and $j = \{1,2,3,4\}$. The resulting 3D coordinates for the position of the vehicle are depicted in Fig. 4.2. It shows the resulting square path according to the references θ_r and ϕ_r.

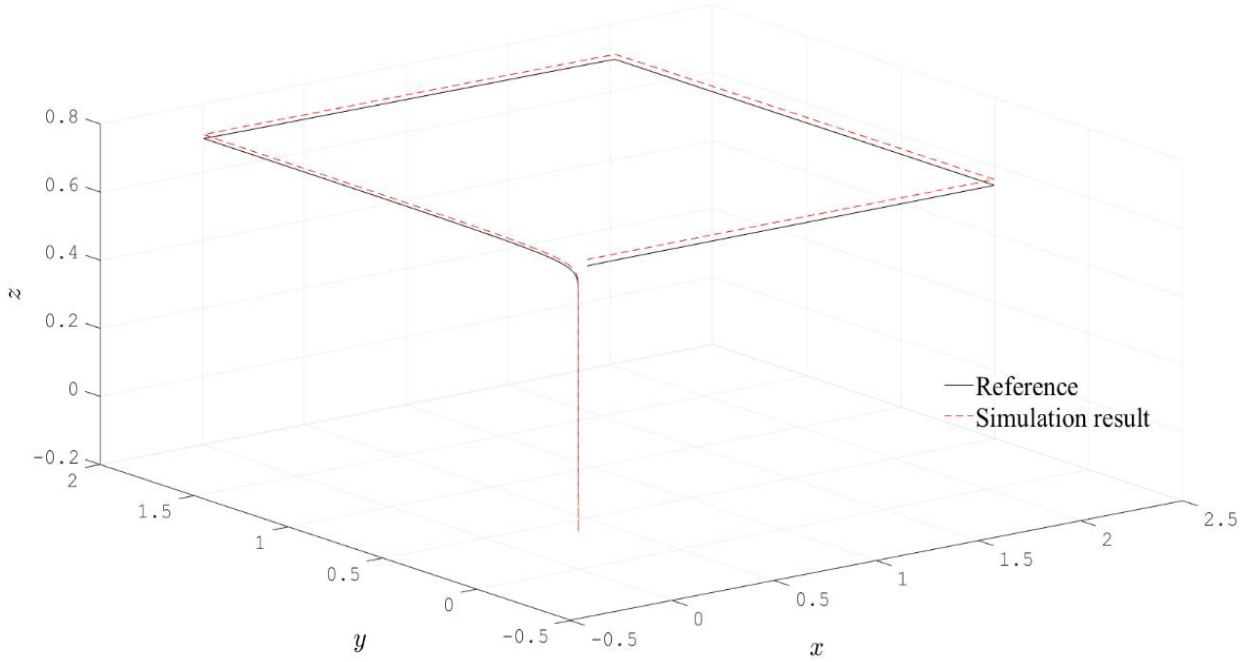


Fig. 4.2 3D output path resulting from the outer loop backstepping controller simulation.

In addition, the resulting Euler angles, corresponding to the attitude of the UAV, are shown in Fig. 4.3. It can be noted that the control objective is fulfilled as the angles converge to their

respective reference in finite time. Also, three yaw overshoots are appreciated, caused by the transitions of the roll and pitch from a constant value to a time variant value, and vice versa.

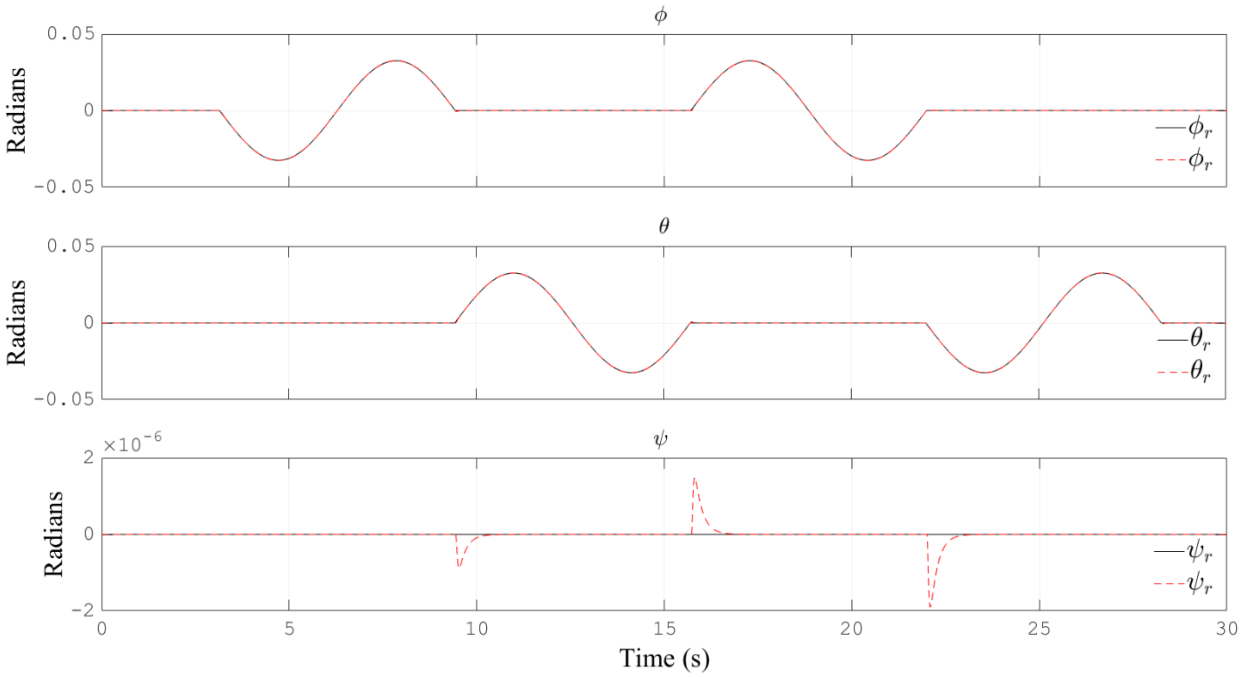


Fig. 4.3 Euler angles and their references resulting from the outer loop backstepping controller simulation.

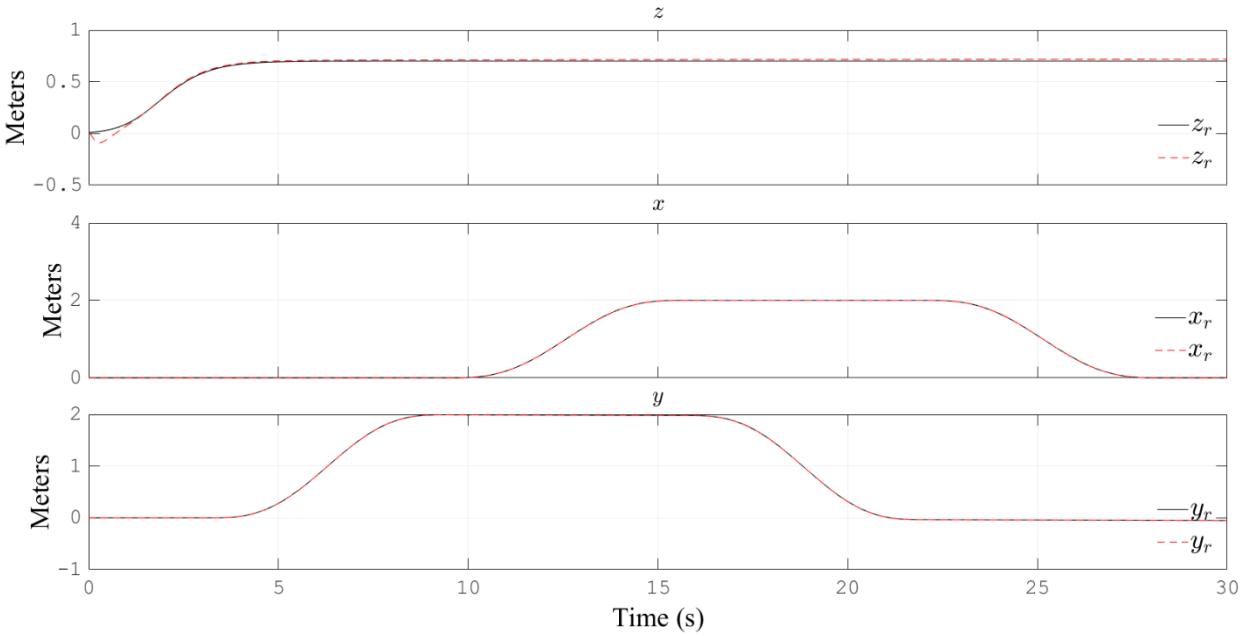


Fig. 4.4 Position coordinates resulting from the outer loop backstepping controller simulation.

Now, Fig. 4.4 shows the position coordinates of the aircraft. As stabilize flight mode is defined, only the altitude z is controlled, which performed satisfactorily. Lastly, Table 4.1 depicts three different indices of the RMSE (root mean square error) corresponding to the reference tracking performance of the experiment. It will be used to evaluate and select one option from the three proposed controllers.

TABLE 4.1. RMSE CALCULATION FOR THE REFERENCE TRACKING SIMULATION OF THE BACKSTEPPING CONTROLLER

Calculus	ϕ	θ	ψ	z
$\Sigma(e_t^2)$	1.46572× 10^{-5} rad^2	1.44753× 10^{-5} rad^2	1.10785× 10^{-10} rad^2	1.120496847 m^2
$\Sigma\left(\frac{e_t^2}{n}\right)$	4.8841× 10^{-9} rad^2	4.82351× 10^{-9} rad^2	3.69162× 10^{-14} rad^2	0.000373374 m^2
$\sqrt{\Sigma\left(\frac{e_t^2}{n}\right)}$	6.98863× 10^{-5} rad	6.94515× 10^{-5} rad	1.92136× 10^{-7} rad	0.019322901 m

4.2.2 Backstepping Sliding Mode Controller

The next simulation results are obtained with the backstepping sliding mode controller. The controller gains used for this simulation experiment are given by

$$k_i = 3, k_{i+1} = 0.05, k_7 = 3, k_8 = 0.3, k_{jM} = 4000 \tag{4-6}$$

for $i = \{1,2,3\}$ and $j = \{1,2,3,4\}$.

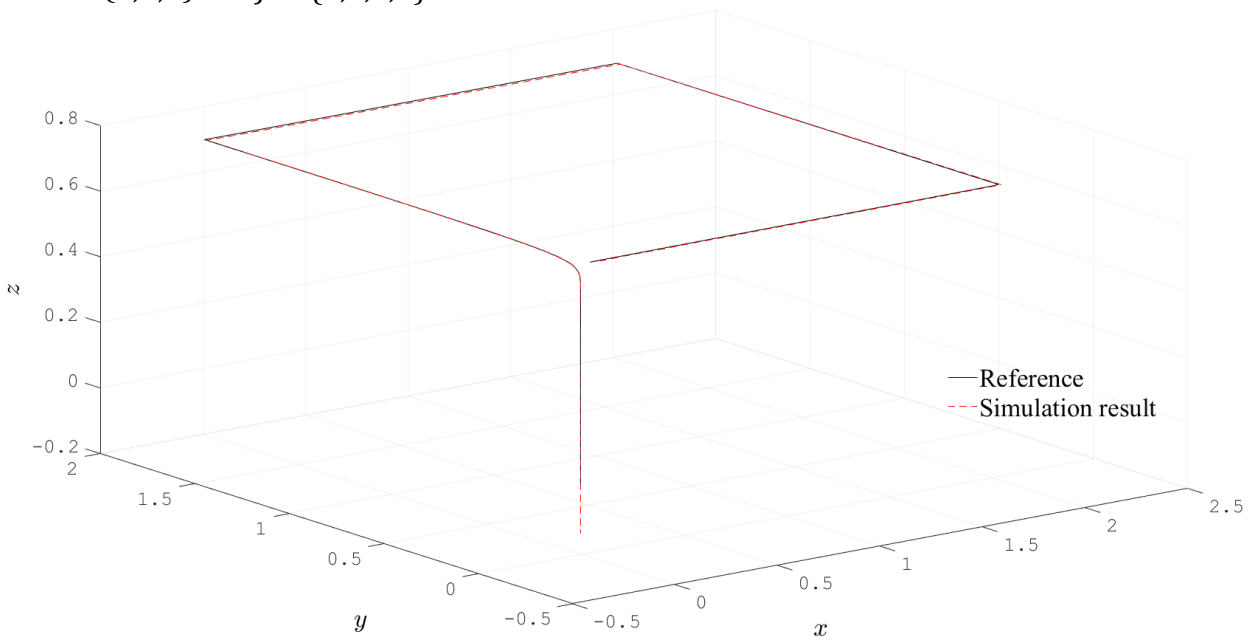


Fig. 4.5 3D output path resulting from the outer loop backstepping sliding mode controller simulation.

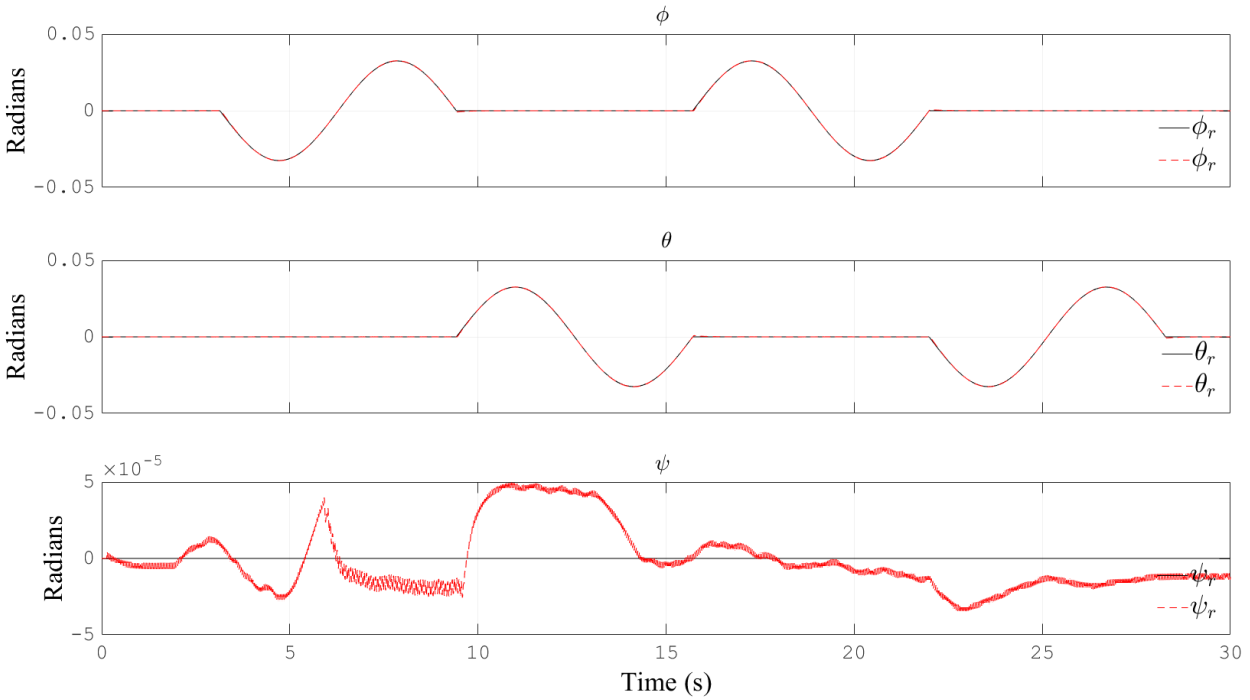


Fig. 4.6 Euler angles and their references resulting from the outer loop backstepping sliding mode controller simulation.

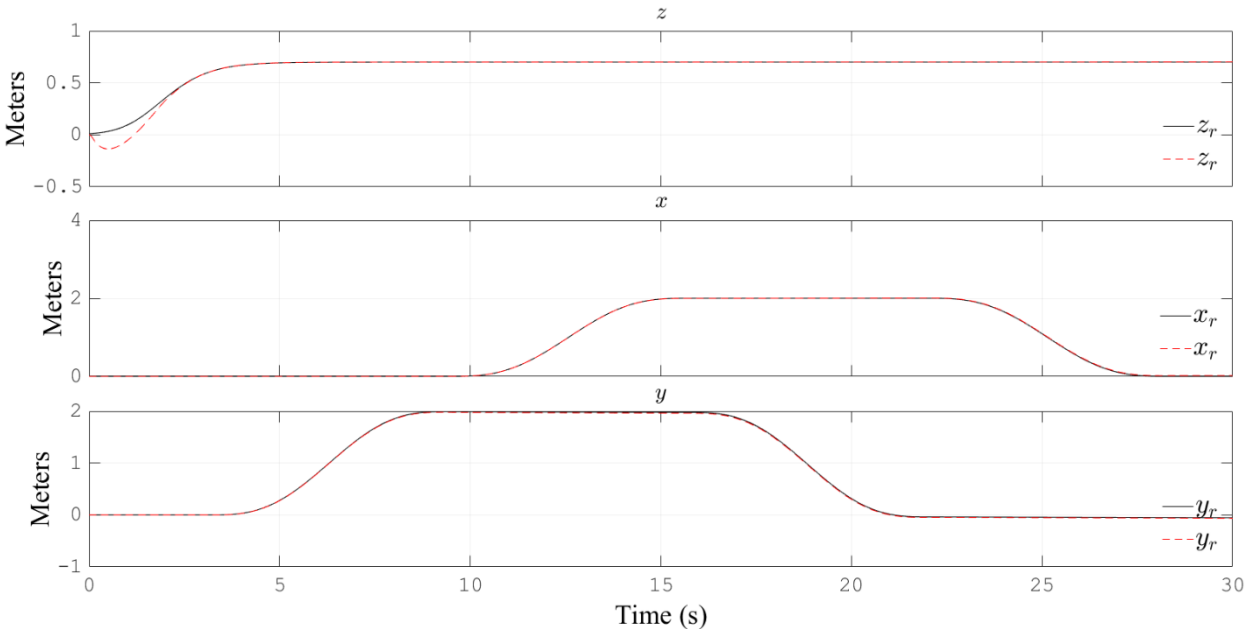


Fig. 4.7 Position coordinates resulting from the outer loop backstepping sliding mode controller simulation.

The resulting 3D coordinates for the position of the vehicle are depicted in Fig. 4.5. It shows the resulting square path according to the references θ_r and ϕ_r. A significant attenuation can be appreciated in the tracking of the variables, which is a direct consequence of the

implementation of the sliding mode term in the control law.

Moreover, the resulting Euler angles, corresponding to the attitude of the vehicle, are depicted in Fig. 4.6. It can be noted that the control objective is fulfilled as the angles converge to their respective reference in finite time. In comparison with the previous controller, the overshoots are eliminated but a high frequency component is appreciated in the response. This is a characteristic of the variable structure control algorithms as sliding modes.

Therefore, Fig. 4.7 shows the position coordinates of the aircraft. As stabilize flight mode is defined, only the altitude z is controlled, which performed satisfactorily. Lastly, Table 4.2 depicts three different indices of the RMSE corresponding to the reference tracking performance of the experiment.

TABLE 4.2. RMSE CALCULATION FOR THE REFERENCE TRACKING SIMULATION OF THE BACKSTEPPING SLIDING MODE CONTROLLER

Calculus	ϕ	θ	ψ	z
$\Sigma(e_t^2)$	5.11457× 10^{-5} rad^2	4.78446× 10^{-5} rad^2	1.24818× 10^{-6} rad^2	2.981129865 m^2
$\Sigma\left(\frac{e_t^2}{n}\right)$	1.70429× 10^{-8} rad^2	1.59429× 10^{-8} rad^2	4.1592× 10^{-10} rad^2	0.000993379 m^2
$\sqrt{\Sigma\left(\frac{e_t^2}{n}\right)}$	0.000130548 rad	0.000126265 rad	2.03941× 10^{-5} rad	0.031517913 m

4.2.3 PID Controller

The last simulation results correspond to the controller based on the PID methodology. The controller gains used for this simulation experiment are given by

$$k_{Pi} = 2, k_{Ii} = 1, k_{Di} = 0.5 \quad \text{for } i = \{2,3,4\}, \tag{4-7}$$

$$k_{P1} = 15, k_{I1} = 14, k_{D1} = 10, k_{jM} = 4000 \quad \text{for } j = \{1,2,3,4\}. \tag{4-8}$$

The resulting 3D coordinates for the position of the vehicle are depicted in Fig. 4.8. It shows the resulting square path according to the references θ_r and ϕ_r. A significant increase of the tracking error can be appreciated with respect to the previous controllers. This lack of robustness is a feature of the PID controller itself.

Furthermore, the resulting Euler angles, corresponding to the attitude of the vehicle, are depicted in Fig. 4.9. It can be noted that the control objective is fulfilled as the angles converge to their respective reference in finite time. In comparison with the previous controller, the overshoots are present again in the yaw response, which is, also, a feature of the PID controller due to the

compromise between a minimum stabilization time and a minimum control signal magnitude. Fig. 4.10 shows the position coordinates of the aircraft. As stabilize flight mode is defined, only the altitude z is controlled, which performed satisfactorily. Finally, Table 4.3 depicts three different indices of the RMSE corresponding to the reference tracking performance of the experiment.

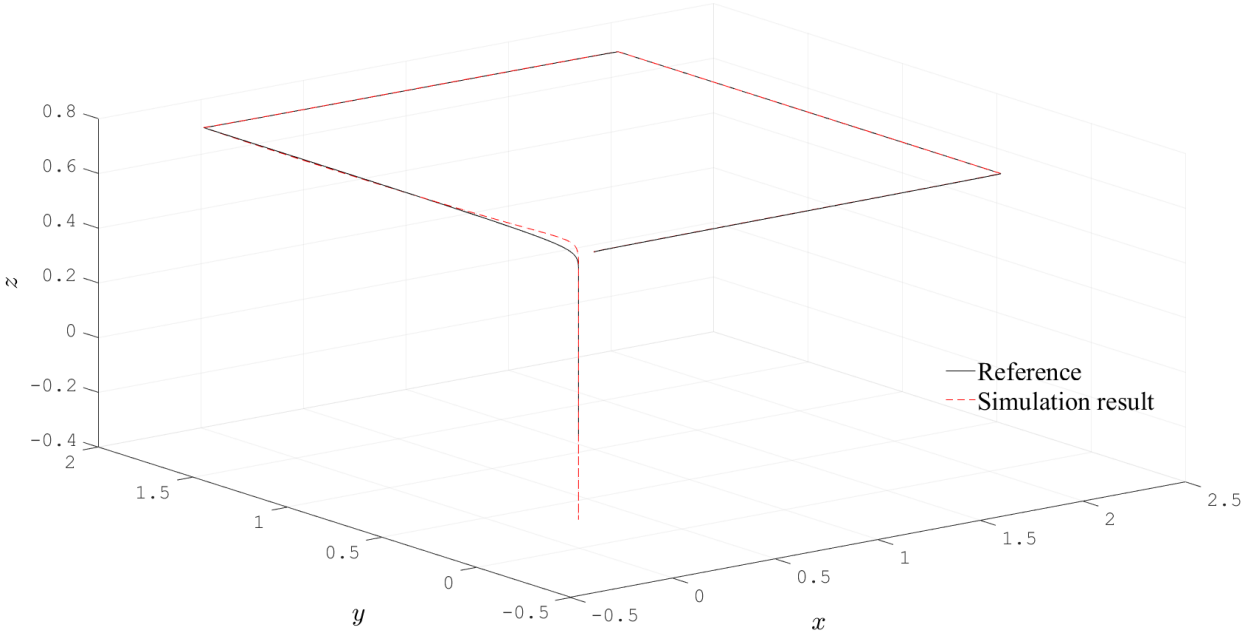


Fig. 4.8 3D output path resulting from the outer loop PID controller simulation.

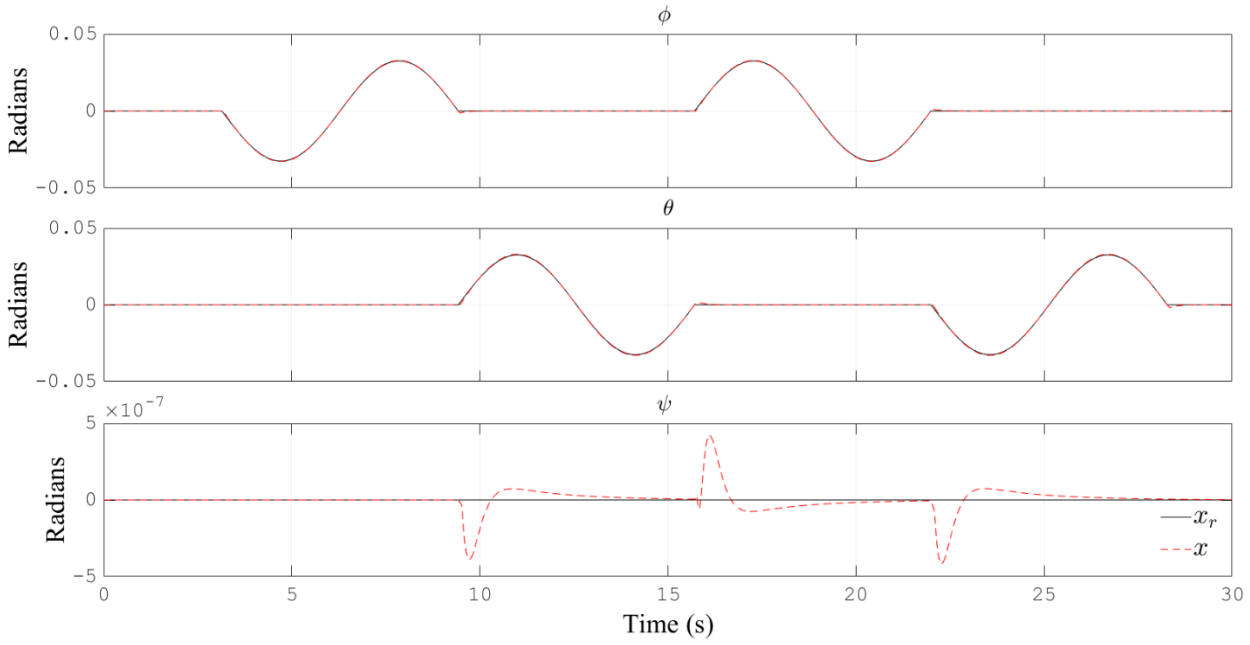


Fig. 4.9 Euler angles and their references resulting from the outer loop PID controller simulation.

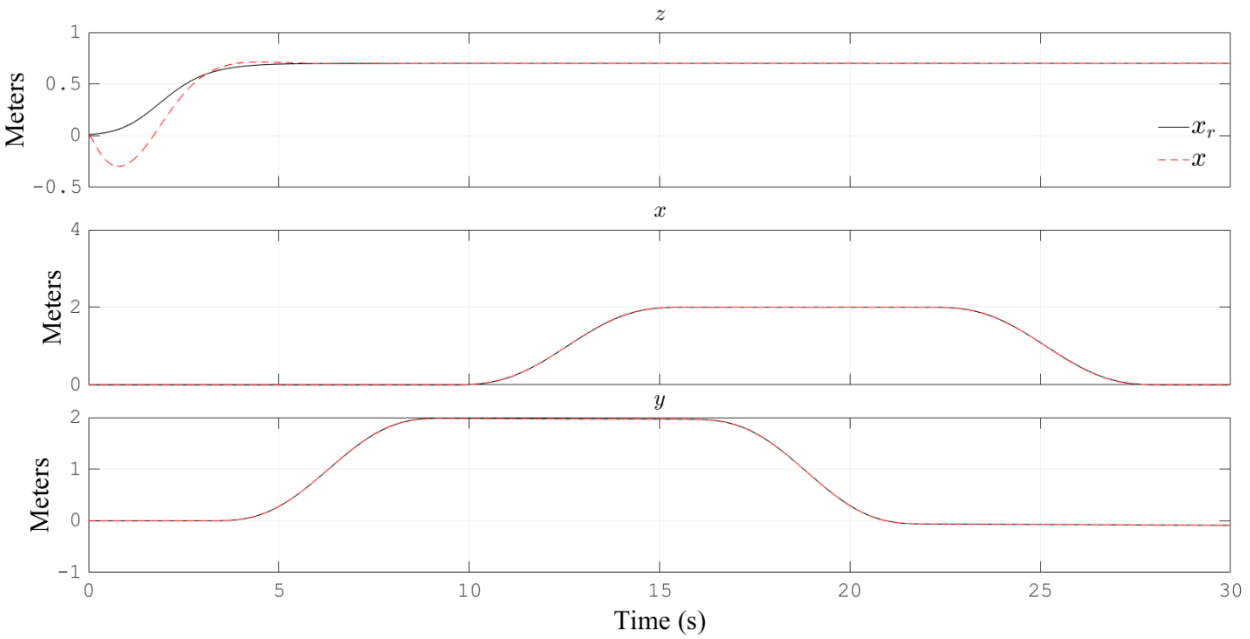


Fig. 4.10 Position coordinates resulting from the outer loop PID controller simulation.

TABLE 4.3. RMSE CALCULATION FOR THE REFERENCE TRACKING SIMULATION OF PID CONTROLLER

Calculus	ϕ	θ	ψ	z
$\Sigma(e_t^2)$	0.000112219 rad^2	0.000262295 rad^2	1.83042× 10^{-11} rad^2	18.08997305 m^2
$\Sigma\left(\frac{e_t^2}{n}\right)$	3.7394× 10^{-8} rad^2	8.74025× 10^{-8} rad^2	6.09936× 10^{-15} rad^2	0.006027982 m^2
$\sqrt{\Sigma\left(\frac{e_t^2}{n}\right)}$	0.000193375 rad	0.000295639 rad	7.80984× 10^{-8} rad	0.077640078 m

From the analysis in Table 4.1, Table 4.2, and Table 4.3, it can be concluded that the best controller performance was demonstrated by the backstepping algorithm based on the lowest error rates generated. Also, the low robustness of the PID controller and the high frequency components in the response of the backstepping sliding mode controller are good arguments to discard them.

4.3. Inner Loop Backstepping Controller

As described in section 3.3, the references ω_{ir} for the inner loop controller are obtained from the control laws U_1, U_2, U_3 and U_4 of the outer loop controller. The tracking of these references is achieved by means of the voltage applied to each of the motors of the vehicle.

It is important to notice that these voltage signals are not directly injected into the motors. Instead, these are converted to duty cycles of PWM signals, which are generated by the embedded

system. The electronic speed controllers (ESC) later convert them to the corresponding voltages that are applied to the motors.

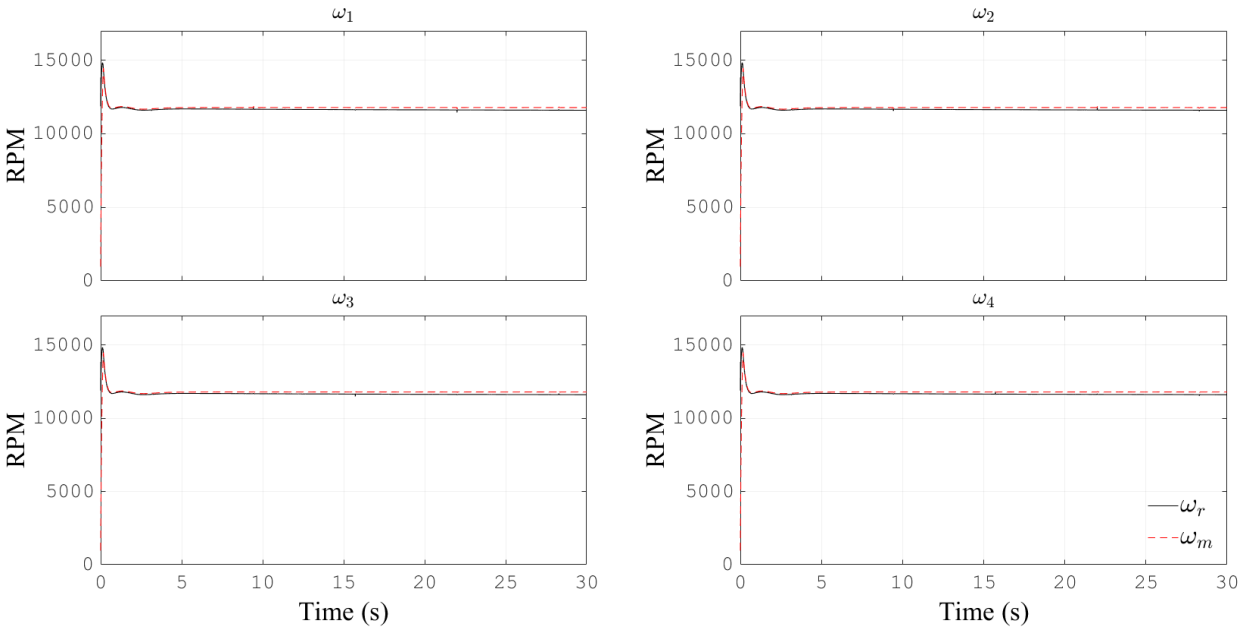


Fig. 4.11 Reference tracking of the angular velocities for the inner loop backstepping controller simulation.

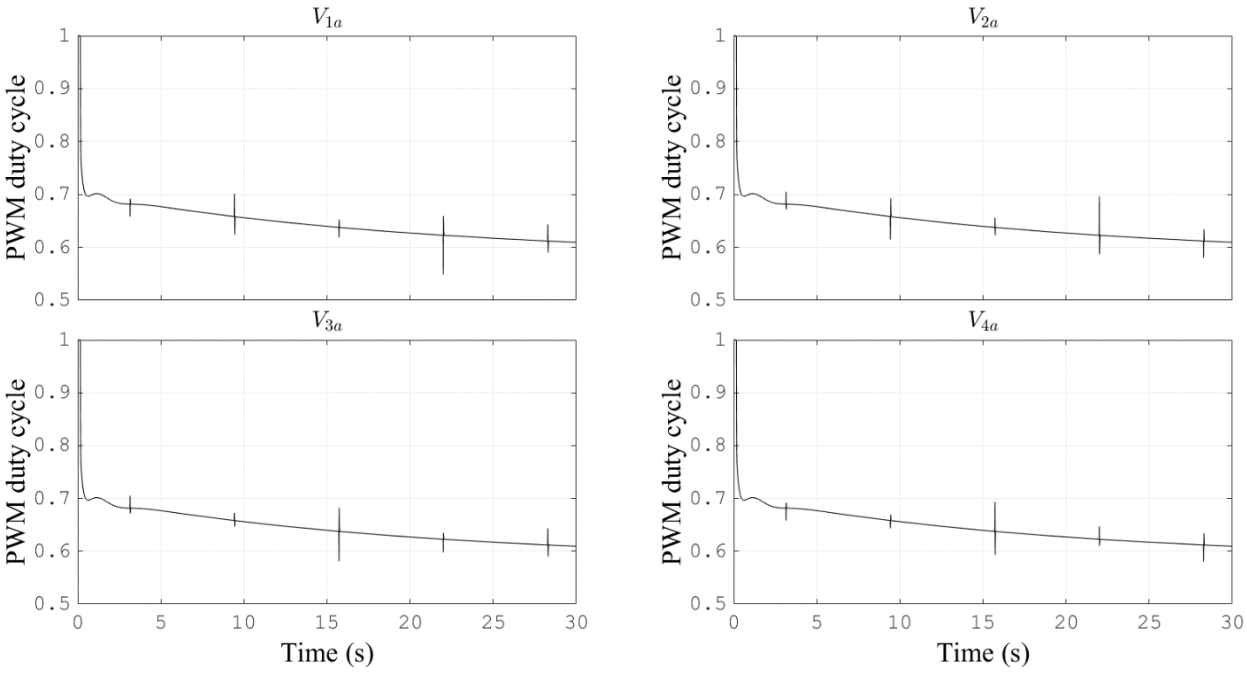


Fig. 4.12 PWM duty cycles corresponding to the voltages generated by the inner loop backstepping controller.

The resulting reference tracking for the four motors of the vehicle are presented in Fig.

4.11. It can be observed that the tracking was achieved successfully, and that the magnitude of the generated angular velocities is within the range of a standard BLDC motor for UAV. Also, the settling time and maximum overshoot for the resulting tracking response are minimal as required for assuring the stabilization of the outer loop controller. Finally, Fig. 4.12 depicts the PWM duty cycle signals corresponding to the voltages generated by the controller.

4.4. Conclusions

Simulation environments are normally defined under ideal situations and, hence, they usually discard factors as disturbances and unmodeled dynamics. In order to evaluate the effectiveness of the proposed control schemes, simulation is a good starting point; nevertheless, the controller needs to be implemented in an embedded system to perform real-time experiments.

Based on the analysis of the simulations results obtained for the three control algorithms for the outer loop, the backstepping controller is selected to develop the real-time experiment. Also, the same technique will be used for the inner control loop.

In the next section, the methodology for the embedded implementation of the proposed control scheme is presented and the issues faced during the process are discussed. Furthermore, the code embedded in the main controller board is developed and explained, aiming to test the aircraft by interconnecting all of its subsystems.

5. Multi-Rotor Embedded System Development

After a satisfactory simulation stage, the next step is to translate the designed controllers from a simulation environment to embedded code for real-time execution. This process is required because Matlab-Simulink uses code that is interpreted, not compiled, and is not compatible with embedded systems. Even though Matlab provides a tool for code generation, it generates several lines of unused code and headers. For that reason, manual code development is performed.

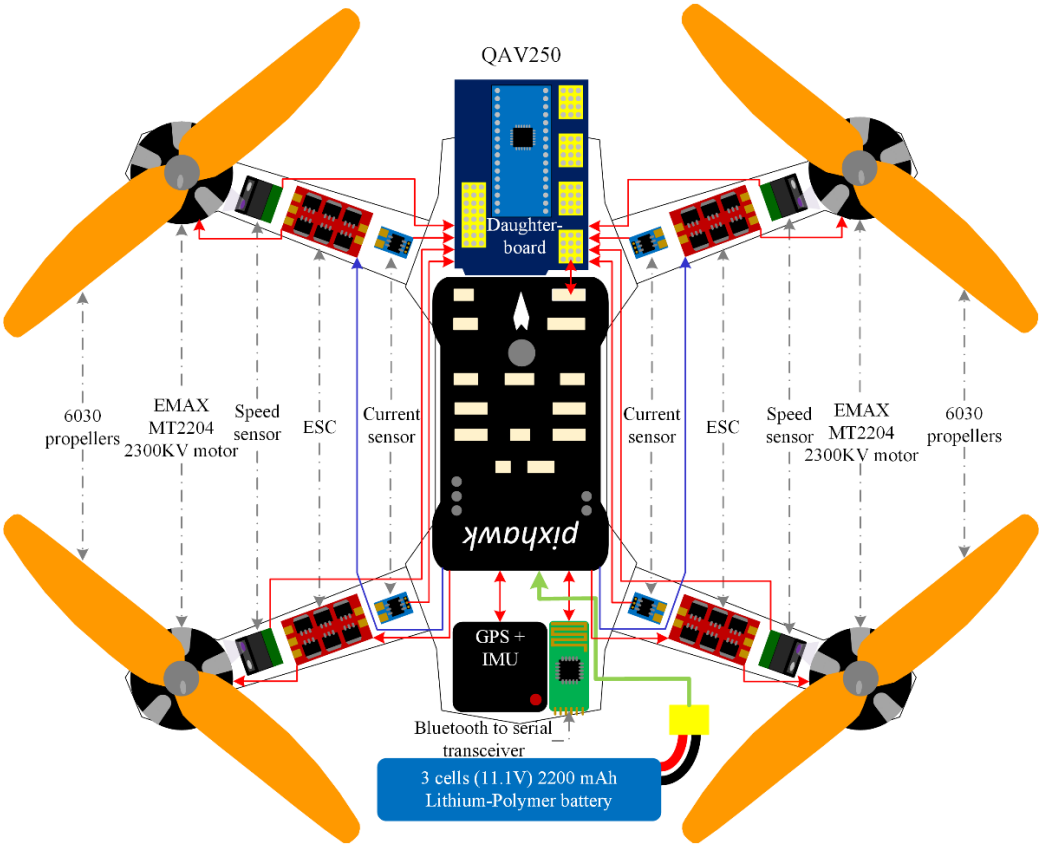


Fig. 5.1 Overview of the UAV components and the electrical interconnections.

As the overall control scheme is based on feedback, it relies on retrieving the status of important variables of the vehicle. For the outer control loop, these variables are its attitude and altitude, as a stabilize flight mode is implemented. For the inner control loop, the variables to retrieve are the current and angular velocities of the BLDC motors. The first phase of this process is carried out by sensors which are interfaced with the embedded system by means of appropriate

ports as analog to digital converter (ADC) inputs or digital inputs. Whereas the main embedded control system has a limited number of inputs, the use of a daughter-board is proposed to receive the measurements from a sub-set of the sensors of the vehicle and to maintain a constant communication with the main embedded board. The main board used for the implementation of the controller is the Pixhawk [PX4DT-18] and the daughter-board is a generic board based on an ATMEGA328/P microcontroller [Atmel-16]. The configuration of the embedded control architecture and its components is presented in Fig. 5.1.

The rest of this chapter presents the role of these boards in the overall embedded controller and their relationships with the sensors and the actuators of the UAV.

5.1. Daughter-Board Embedded Implementation

This board is dedicated to interface the currents sensors and speed sensors of the actuators with the inner loop control algorithm, which is implemented in the main board. Moreover, the conditioning of the signals retrieved from the sensors is developed in the daughter-board as well. For instance, filtering stages are implemented in this board to improve the characteristics of the noisy measurements received by the sensors at a high-speed frequency. The communication between the daughter and the main board is stablished by using a serial protocol and data request polling. The daughter-board is based on an ATMEGA328/P microcontroller, which is capable of running at a frequency of 16 MHz, has 8 ADC channels available with up to 10-bit precision, and a universal synchronous and asynchronous receiver-transmitter (USART). As a result, the board can be used to sample up to 8 analog sensors and transmit the data through serial communication to the main board.

5.1.1 Motors Current Sensors

The electronic speed controllers (ESC) used in the UAV can support up to 12A when driving the brushless motors. Therefore, the same current rate, or higher, needs to be used to measure the power consumption in the motors during flight. For that, ACS712ELCTR-20A-T devices are used, which are low profile SOIC8 current sensors that can be attached to the UAV, without compromising the weight load that it can support. Basically, the sensors are linear Hall

Effect devices with copper conduction path that generates a magnetic field that is sensed by the Hall integrated circuit (IC) and converted to a proportional voltage signal. The device has a response time of 5 μs and can operate with a 5V supply voltage, which is suitable for embedded systems and fits in the UAV circuitry. Moreover, due to the power saving needs of UAVs, these devices have an internal conductive resistance of 1.2 Ω that provides low power loss and supports up to 5 times the overcurrent conditions.

Furthermore, based on Fig. 5.2, there is no need for using signal isolators as far as sensor pins 5 through 8 are isolated from the conductive paths. The sensors can read the current in both directions, from I_p+ to I_p- and vice versa. The device specification provides a chart that involves both cases, where can be observed that there is an offset which identifies where the current direction transition occurs, given the following equation

$$V_{\text{sensor-offset}} = \frac{V_{cc}}{2} \tag{5-1}$$

where V_{cc} is the supply voltage for the integrated circuit, defined as 5V.

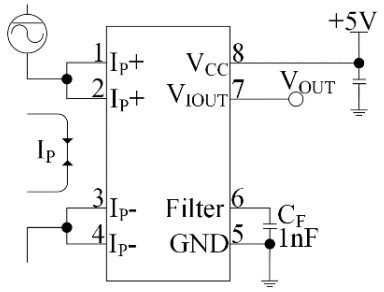


Fig. 5.2 Typical application of ACS712.

Hence, using the sensitivity of the sensors retrieved from [Allegro-17], the equation for the measured current can be defined as

$$V_{out} = V_{\text{sensor-offset}} + 0.1i \tag{5-2}$$

$$ADC_{read} = \frac{1023}{V_{cc}} V_{out} \tag{5-3}$$

$$ADC_{read} = \frac{1023}{V_{cc}} \left(\frac{V_{cc}}{2} + 0.1i \right) \tag{5-4}$$

$$i = \frac{\frac{ADC_{read} V_{cc}}{1023} - \frac{V_{cc}}{2}}{0.1} = 10V_{cc} \left(\frac{ADC_{read}}{1023} - \frac{1}{2} \right) = 50 \left(\frac{ADC_{read}}{1023} - \frac{1}{2} \right) \tag{5-5}$$

where ADC_{read} is the value obtained by the daughter-board by means of a 10-bit precision ADC.

Seeing that the maximum and minimum values that the sensor can return are 4.5V (920 in

the ADC read) and 0.5V (102 in the ADC read), respectively, the maximum and minimum currents that can be obtained are 20A and -20A. With the goal to avoid decimals in the calculation due to embedded system floating-point limitations, the ADC_{read} parameter is multiplied by a factor of 1000, meaning that the $\frac{1}{2}$ in the equation is seen as 500, as follows

$$i_{embedded} = 50\left(\frac{1000ADC_{read}}{1023} - 500\right). \tag{5-6}$$

Given (5-6), the current is expressed in milliamps format to be able to have more granularity in the embedded calculation.

5.1.1.1 ADC Data Capture and Transmission

The communication between the main and daughter-boards permits to the motors control algorithm, which is implemented in the main board, to define the necessary feedback from the current sensors. To that end, the daughter-board is configured to interrupt whenever an ADC conversion is ready and manually triggers the ADC for the next channel to be sensed. Basically, the board does a sweep for all the channels defined in a macro at a maximum of 8, and after reaching that limit, the ADC channel setting starts again from the first port. The implementation of these functionalities are described in the code depicted in Fig. F.1 and Fig. F.2 of the appendices. Hence, the daughter-board continually reads the sensors and sends the data through serial protocol when requested by the motors control algorithm implemented in the main board.

It is important to notice that a normal conversion takes 1.5 ADC clock cycles for sample-and-hold after setting the start of conversion bit (ADSC), and 13 clock cycles later, the conversion completes [Atmel-16]; after that, a 1 cycle period is needed in order to change the ADC channel. That gives a total of 15.5 ADC clock cycles for a single conversion to be ready, thereby, 124 ADC cycles are needed, for a maximum of 8 channels, as obtained in the following equation

$$\text{ADC\_cycles\_for\_8CH} = 8(1.5 + 13 + 1) = 8(15.5) = 124 \text{ clock cycles} \tag{5-7}$$

where the clock cycle is the ADC clock defined in the ADCSRA register, which is 2 MHz. In that regard, the time required to complete all the conversions is 15.5 μs. Nevertheless, a single conversion is not accurate enough for measuring the motor current, and after inspection, it was noticed that 50 samples are a reasonable number that provides good accuracy, based on measurements. In this case, after all the operations required, the measurement provides a total of 5.24-ms for all 8 channels and 2.63-ms in the case of 4 channels.

Once the sensor values are captured, the next step is to define a routine for capturing the requests from the main board. In this context, the USART device is configured in asynchronous mode on the daughter-board for sharing the ADC and speed captures. The implementation requires being ready whenever a data request arrives, and for that purpose, two different methods are defined. The first receives interrupts configured to respond as soon as a data request is received, as coded in Fig. F.3, and another for sending data once a board pin voltage level change is detected, as coded in Fig. F.4. These requests tell the daughter-board that the UAV controller board requires data, therefore the secondary board responds with a package containing the information. See Fig. F.5 for references on USART interrupt routine and Fig. F.6 for references in voltage level change interrupt routine.

As far as the data requests is received at a 100 Hz frequency (10-ms time period) from the UAV controller scheduler, the transmission cannot exceed that lapse. Hence, the UART is defined to run at 500,000 bits per second (bps), as described in Fig. F.3, which gives the ability to send 625 bytes in the scheduler period of the selected routine for getting the data. That gives enough space for sending the ADC and speed values and other information that might be required later, which is obtained as

$$\text{Number\_of\_bytes} = \frac{\left(\frac{bps}{F}\right)}{\text{Bits\_in\_a\_byte}} = \frac{\left(\frac{500000}{100}\right)}{8} = 625 \tag{5-8}$$

where F is the UAV controller scheduler routine frequency, and bps is the baud rate [Atmel-16].

It is important to remark that during debug, it was found that because of the latency of the interrupt functions and data transmission, both were getting interrupted by the ADC interrupt vector, causing corruption of the ADC conversion array. Therefore, ADC interrupt disabling is required during the transmission process, and the subsequently interrupt restoration at the end of the data transference. The code that exemplifies the disabling and enabling of the ADC interruptions is shown in Fig. F.7 and Fig. F.8, respectively, meanwhile the code for transmission is shown in Fig. F.9.

5.1.1.2 Motor Current Sensor Configuration and Validation

After that the current sensing mechanism is set up, the next step is to test the functionality by running the brushless motors at different speeds, and by capturing the ADC reads from the ACS712. The current that is consumed by the motor needs to flow through the sensor, and for that

purpose, the circuit interconnection for testing is shown in Fig. 5.3.

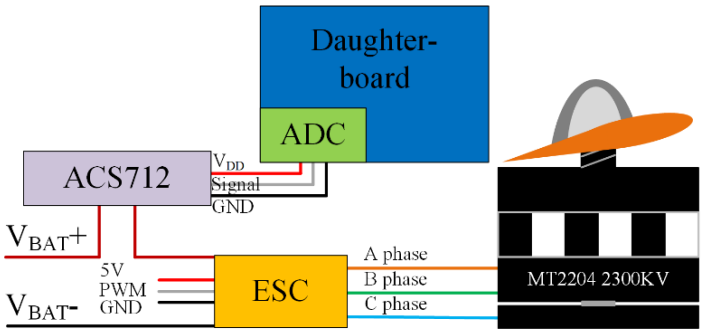


Fig. 5.3 ACS712 sensor interconnection.

The measuring was performed by using different throttle settings and by sending the read data through serial communication to a host machine. Numerous reads are captured for each throttle value from 1000 to 2000, in increments of 50. During testing execution, noticeably sensor noise response was observed causing several ADC capturing bits fluctuation.

The propellers considered for the testing are the 6030, which are 1 inch larger than the default 5030 and provide more thrust force, but as a sequence of the different inertia element and weight of this propellers specification, the speed capabilities are reduced and more power consumption is generated when used with the selected MT2204 2300KV brushless motors.

With the aid of an oscilloscope, the sensor output signal was captured at the same time that the ADC was converting the signal, and by observation, it was noticed that the signal trends at the mean of the fluctuation, that gives the advice of the need of a filtering mechanism for having better performing reads, and that is the main reason for coding the multiple ADC sampling [Kester-06]. In Fig. 5.4, the current consumption can be observed, which is calculated by using the mean of the ADC captures regarding the previous explanation.

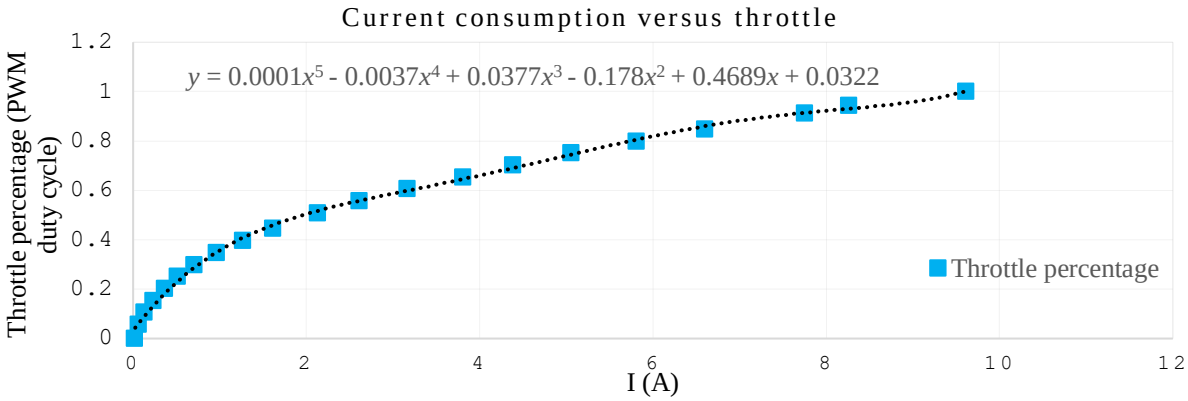


Fig. 5.4 Current to PWM duty cycle characteristic curve.

5.1.2 Motor Speed Sensors

The objective of the motors control algorithm, which is based on the backstepping technique, is to control the angular velocity ω of each actuator for tracking the references ω_r defined by the outer control loop. Since the BLDC motor used in the UAV considered in this work do not have Hall effect sensors or an encoder coupled to its axle, an intermittent black and white band is attached in the motor case and CNY70 reflective sensors are used for capturing the light intensity change in the band when the motor is rotating, generating a pulse frequency that feeds a pin voltage level change interrupt in the daughter-board, as implemented in [González-Hernández-12]. This represents an alternative of an optical encoder which is simple and low cost.

The interruption generated by the encoder is time tracked with a timer, designated in the daughter microcontroller for that specific purpose and five different captures are made. The first capture takes a false pulse, which is taken in the middle of a pulse generated (not at the rising or falling edge), and then the next four pulses are timer captured, having two complete periods observed, which are averaged to obtain better accuracy. The black and white band of the encoder generates 40 level changes within a motor revolution, as shown in Fig. 5.5, and the reason to have multiple trackers is the time it takes to the motor to go through a single revolution, which is high regarding the timing limitation of 10 milliseconds for capturing the sensor in the daughter-board, sending the data to the main controller, and calculating the control outputs. Therefore, the more tags are in the encoder band, the less it takes to know the motor speed if only a couple of pin voltage level changes are time measured.

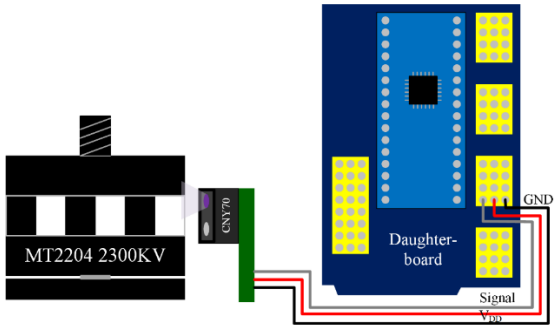


Fig. 5.5 CNY70 reflective sensor setting and interconnection with the daughter-board.

The code that takes care of initializing the timer for time capturing is shown in Fig. F.10, while the initialization for the pin voltage level change ports and their interruptions are displayed in Fig. F.4 and Fig. F.11, respectively.

It is important to notice that the code for current sensing from section 5.1.1 is running all together with the motor speed sensing algorithm, which can cause interruption overlap if they are not handled properly. For resolving that problem, an interrupt scheduler routine is created to assign execution status to the current and speed sensing tasks. In that way, only one interruption is running at a time, avoiding the interrupt miss for the speed period capturing, which is extremely time sensitive. The code for that implementation is shown in Fig. F.12. The results of the speed measurements at various throttle values are displayed in Fig. 5.6.

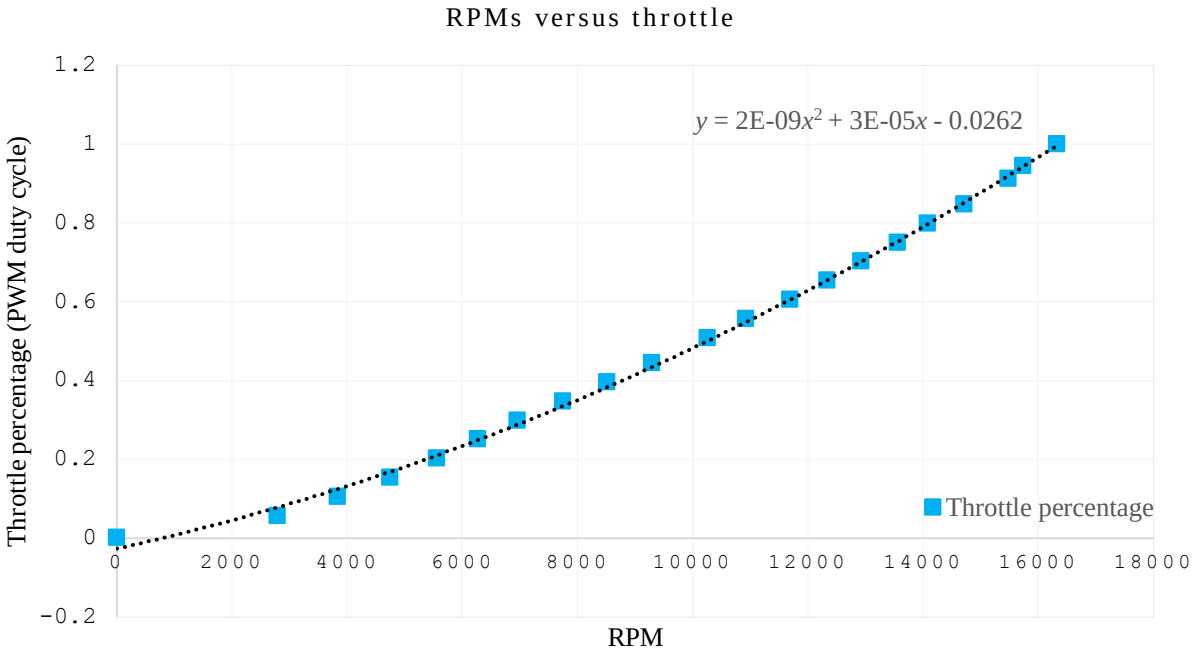


Fig. 5.6 RPM speed to PWM duty cycle trending function.

5.1.2.1 Motor Speed Sensing Filter

During the motor control testing, it was noticed that the measurements from the encoder present fluctuations that inject errors in the UAV controller calculation. Therefore, the necessity of a digital filter is evident and two different schemes are implemented. The first one is a moving average (MA) filter, which considers 11 samples according to the following equation

$$\mathrm{MA}_n = \frac{\sum_{i=m}^{n} x_i}{n-m} \tag{5-9}$$

where n is the current sample number, m is n minus the number of samples desired for the average calculation, and x_i is the ith raw data sample. The second filter approach is an exponentially weighted moving average (EWMA), where successively declining weights are applied as the

calculation goes further back in history as defined in

$$\mathrm{EWMA}_n = (1-\alpha)\mathrm{EWMA}_{n-1} + \alpha x_n = \mathrm{EWMA}_{n-1} + \alpha(x_n - \mathrm{EWMA}_{n-1}) \quad (5\text{-}10)$$

where α is the weight constant for the raw samples data, which presents a minimal error with a value of 0.132.

In the analysis of both filters, the EWMA exposes a smaller RMSE, which is the acceptance criteria considered for the filter to be chosen. Moreover, EWMA provides smoother means of averaging, where data becomes gradually less influential as it ages, while MA treats each of the observations in the calculation equally when talking about importance, and later are suddenly disregarded as soon as the samples fall off the end of the number of elements considered [Everett-11]. The code for the filter is shown in Fig. D.23.

5.1.3 Graphical User Interface (GUI) for Data Capturing

The system's response can be obtained by exciting the motor with a step signal for a certain period and by sensing the current consumed by the motor during the time interval of the impulse. This is a standard procedure for obtaining the transfer function of a single input-single output (SISO) unknown system. For that purpose, a 100% duty cycle is injected into the ESC PWM input. During the test, serial communication is used for obtaining the data about current consumption and speed, but due to the lack of a data container in the microcontroller for the complete test, a graphical user interface (GUI) is needed for logging all the information, show the behavior in a graph, and write a data file containing all sensed variables.

The GUI was created by using Java[6] code due to its number of classes already available and for portability. Wireless serial communication is required and for that purpose, thus, a Bluetooth communication device is mounted in the aircraft for sending the data to the GUI. The UART protocol rate is defined at 921,600 bits per second (BPS), as supported by the wireless device, while the Pixhawk board defines two different baud rates, one for the GUI and another for the daughter-board connection; the last defined at the same rate indicated in 5.1.1. The code for that implementation is shown in Fig. E.1 Notice that an action listener is added in the code for tracking the serial connection status, which opens or closes the UART connection and also finishes

[6] Java, Version 1.8.0_162, Oracle Corporation, 500 Oracle Parkway, Redwood Shores, CA 94065, 2018.

the execution of the sensors tracking. Due to the time restrictions to execute the controller code, the data transmission needs to be optimized as much as possible, which ends on defining a high transmission rate as mentioned. Since the algorithm needs to read data whenever it comes, a thread is required to oversee that task. Therefore, serial communication receiving thread is created, as shown in Fig. E.2. In this figure, it can be noticed that another routine is being called (***PrintGraph***); that function is in charge of capturing the serial data and split it into the multiple messages that the serial line received contains, also performing a caching method for better graphical performance while displaying the data in multiple panes. Fig. E.3 shows the algorithm of its implementation.

Furthermore, a handler function for generating signals for the data request was identified as required, in order to emulate the UAV controller scheduler, which executes every 10-milliseconds. At the end of the execution, when the serial communication is finished, all the data is logged in a comma-separated values (CSV) format file to be later used for system identification and flying parameters analysis. The code that executes that task is defined in Fig. E.4. A description of the designed GUI and its elements is depicted in Fig. 5.7.

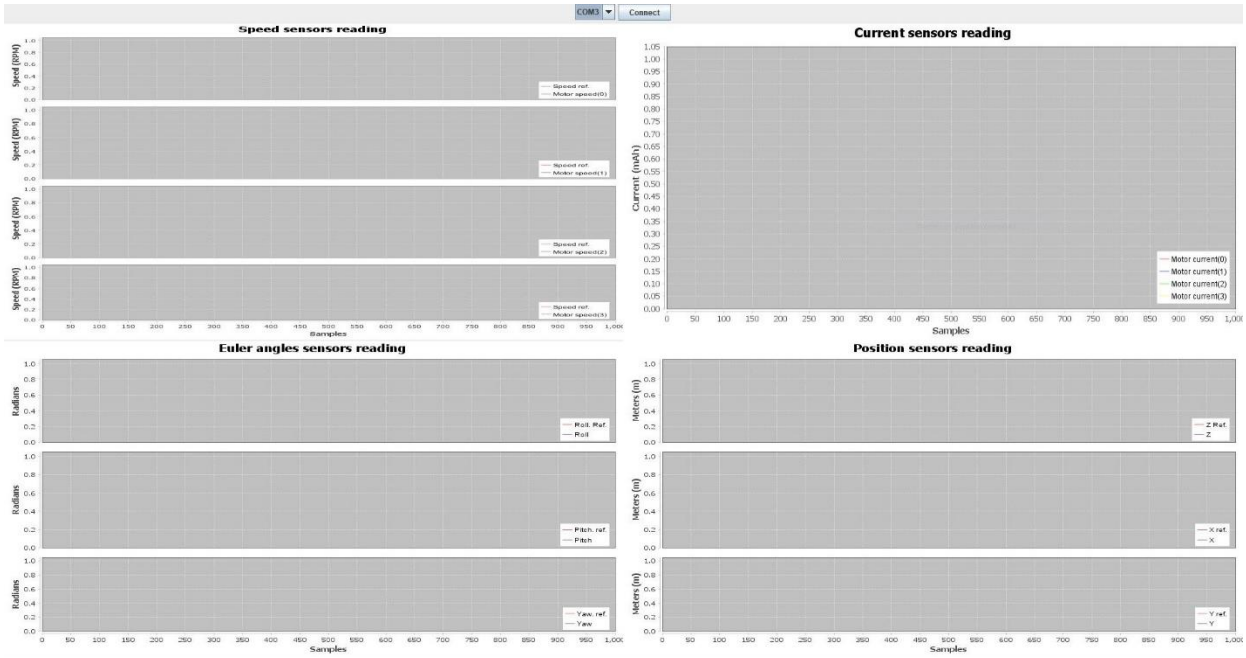


Fig. 5.7 Graphical user interface and its elements.

5.2. Main Board Embedded Implementation

The measurements acquired and conditioned by the daughter-board are received by the

main board to close the outer and the inner control loops. Hence, the control algorithm for both loops are implemented in the main board based on the backstepping algorithm. In addition, the main board directly receives the measurements of the inertial measurement unit (IMU) which provides the current attitude of the vehicle during the experiments.

The implementation of the controllers is developed in a Pixhawk board, which is embedded in the vehicle, for real-time execution. Multiple implications about implementing integrals and derivatives calculation, errors computing, filtering mechanisms, variables value persistence within a function, among others, need to be considered. The following subsections describe the solutions deployed for each of these functions from an embedded C code perspective.

5.2.1 Attitude and Position Sensors

For the outer control loop implementation, it is necessary to retrieve the position coordinates x, y, and z of the vehicle, and its attitude defined by the Euler angles roll (ϕ), pitch (θ) and yaw (ψ). For that reason, sensors of estimators for these variables are required for calculating the error between the reference signals and the vehicle position and orientation in time. The Pixhawk platform, which is the main board in the considered vehicle, has an inertial measurement unit (IMU) that can provide the angles, and also an embedded barometer that can give an approximation of the altitude. However, after testing the barometer, it was noticed that its performance is characterized by relatively large fluctuations on its output, resulting in inaccurate measurement values. Therefore, the proposed solution is to incorporate the MB1043 HRLV-EZ4 sonar sensor which provides millimeter-resolution the z position. This board is connected directly to the Pixhawk using I^2C protocol [Honegger-13], [Dronecode-18]. As the sonar sensor attached in the PX4Flow board provides altitude values in the range from 30 cm. to 5 m. having a blind spot in the 0 to 30 cm. range, a VL53L0XV2 laser sensor is connected in a I^2C splitter and added in the devices list by setting the parameters required under the full parameter list of Mission Planner (RNGFND_TYPE = 16 “VL53L0X”, RNGFND_ADDR = 41, RNGFND_SCALING = 1, RNGFND_MIN_CM = 5, RNGFND_MAX_CM = 120, RNGFND_GNDCLEAR = 10 [ADT-19b]) for providing the 0 to 1 m altitude range value. The MB1043 HRLV-EZ4 sonar sensor is connected in the analog to digital converter (ADC) port of the Pixhawk platform for the larger range, which is sensed and decoded within the ArduCopter code as the Pixhawk only supports one

main altitude sensor. ArduCopter 3.5.7 does not support the VL53L0XV2 sensor natively until version 3.6.0, therefore, the sensor capability was migrated from the later version to the one being used to have the short distance readings for z position control.

Normally, most unmanned aerial vehicles (UAVs) rely on GPS for navigation. However, this navigation solution does not function properly for applications such as planetary exploration, indoor navigation, or under GPS spoofing scenarios. To solve that problem, the optical flow mechanism has been studied. This is a natural solution for navigation that is used by insects like honeybees, dragonflies, and flying birds. It can be treated as a 2D projection of the 3D perceived motion of objects, and it has a wide application for motion estimation [Chao-13]. Various navigation subtasks can be accomplished with the use of optical flow, such as distance estimation altitude hold, obstacle avoidance, velocity and height estimations, and vertical landing. To support of the previously mentioned estimation and autonomous functions, both new vision systems, and novel optical flow motion models are required. The most used devices for generating optical flow are optical computer mouse chips and CCD/CMOS sensors. It works by sensing the apparent motion of the brightness patterns of the feature points in an image, which is a projection of the 3D relative motion into the 2D image plane. The motion can be calculated from the movement of pixels with the same brightness value between two consequent images. For any point (μ, ν) on the image plane, the optical flow vector can be expressed as $[\dot{\mu}, \dot{\nu}]^T$, in the unit of pixels per frame/second, or radian per frame/second, or it can be expressed as a function of image pixels, $f(\mu, \nu)$. A board called PX4Flow [ADT-16b] is an optical flow sensor that can provide information about the position of the drone body frame. It is based on a CMOS machine vision sensor and an ARM Cortex M4 microcontroller that calculates the optical flow in real-time at 250 Hz, however, the update of the optical flow is done at 200 Hz within the Pixhawk code. By using vision and an incorporated 3-axis gyroscope, the sensor obtains the ground texture and visible features to calculate the aircraft ground velocity. With that information, an approximation of the x and y position can be obtained.

During UAV control testing, it was noticed that the compass (yaw angle sensor) readings presented errors while increasing the speed of the rotors. This is due to the electric field produced by the rotors that interfere with the sensing mechanism of the device. To reduce the error, the Mission Planner software allows the user to calibrate the compass while the speed of the rotors is increased to consider the electric field in the process. The propellers are attached backward in the

motors to produce inverted force while keeping the UAV in place, during the increase of speed in steps in the calibration process. The global positioning system (GPS) sensor device attached in the aircraft provides a second compass, and it is enabled in the settings to have measurement redundancy to obtain more accurate sensing. To enable it the second compass, connect the Pixhawk board to the computer, access Mission Planner→Initial setup→Mandatory Hardware→Compass and checkbox "Use this compass" for compass #1 and #2 [ADT-19a].

5.2.2 Firmware Architecture and Execution

As mentioned before, the outer control loop involves diverse subroutines to compute the terms that composes the control terms for the vehicle. In that regard, Fig. 5.8 shows a pseudocode which corresponds to the sequential execution of these subroutines.

```
Define the system and controller constant parameters;
while (Receive UAV flight mode reference signals X_r) do
    Calculate the 1st and 2nd derivatives of the reference signals;
    Retrieve UAV states X for the specific flight mode;
    Filter z measurement with EWMA method;
    Calculate the 1st and 2nd derivatives of the UAV states;
    Compute outer loop controller to obtain ω_r;
    Calculate the derivatives ω̇_r and ω̈_r;
    Retrieve motors velocities ω;
    Filter ω measurements with EWMA method;
    Compute inner loop controller to obtain V_a1−an;
    Transform V_a1−an to PWM_1−n;
    Set PWM_1−n signals in the main board output ports;
end
```

Fig. 5.8 Overall controller execution pseudo-code algorithm.

First, it is necessary to define the constant parameters of the vehicle and both control algorithms. This subroutine is only executed once and is important to do so for improving execution time. Its code is defined in Fig. D.8. Whereas the function works for both flight modes, stabilize and trajectory tracking, two different sets of gains are defined, but for exemplification, only one set is presented in Fig. D.8.

Next in line is the definition of the inverse matrix of the matrix of forces M. As mentioned in (1-6), this matrix is required to obtain the speed to be used as a reference for the motors controller (inner loop). The implementation requires the adjoint and the determinant for a square matrix or the Moore-Penrose pseudo-inverse for a non-square, which involves the transpose of the matrix, multiplications and inverse matrices [MacAusland-14]. Both equations are defined as

$$\text{If } m = n \therefore M^{-1} = \frac{\text{adj}(M)}{\det(M)} \tag{5-11}$$

$$\text{If } m \neq n \therefore \begin{cases} m > n \therefore M^{+} = (M^{\mathrm{T}}M)^{-1}M^{\mathrm{T}} \\ m < n \therefore M^{+} = M^{\mathrm{T}}(MM^{\mathrm{T}})^{-1} \end{cases} \tag{5-12}$$

where m is the number of rows, and n is the number of columns in the matrix. In this implementation, a quad-rotor is being considered, and (5-13) is used. Hence, the definition of the determinant for a matrix of 4×4 can be obtained with

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} \tag{5-13}$$

$$\det(A) = a_{11}|A_{11}| - a_{12}|A_{12}| + a_{13}|A_{13}| - \cdots + a_{1n}|A_{1n}| = \sum_{k=1}^{n} (-1)^{k+1}\, a_{1k}A_{1k} \tag{5-14}$$

where $|A_{1x}|$ is the determinant resulting from subtracting row 1 and column x from the matrix [Grossman-12], forming a 3×3 matrix for the quad-rotor case. For this sub-matrix, the determinant can be obtained as follows

$$|A_{3\times3}| = a_{11}(a_{22}a_{33} - a_{23}a_{32}) - a_{12}(a_{21}a_{33} - a_{23}a_{32}) + a_{13}(a_{21}a_{32} - a_{22}a_{31}). \tag{5-15}$$

With (5-14) as a reference, the code for obtaining the determinant is defined in Fig. D.9, where ***CalcDet4x4Element*** is defined in Fig. D.10 and the determinants of each 3×3 matrix are calculated using (5-15), as coded in Fig. D.11.

The determinant is defined and the next element to resolve is the adjoint of the matrix, which is formally defined as

$$\text{adj}(A) = B^{\mathrm{T}} = \begin{pmatrix} |A_{11}| & |A_{12}| & \cdots & |A_{1n}| \\ |A_{21}| & |A_{22}| & \cdots & |A_{2n}| \\ \vdots & \vdots & & \vdots \\ |A_{n1}| & |A_{n2}| & \cdots & |A_{nn}| \end{pmatrix}^{\mathrm{T}} \tag{5-16}$$

where B^{T} is the transpose of the matrix of cofactors of A. With (5-16) as a reference, the code for calculating the cofactor matrix is defined in Fig. D.12. After that, the transpose is required, and for that purpose, the function in Fig. D.13 is created. The function that computes both tasks is `CalcAdjMatrix4x4` in Fig. D.14, and as its name indicates, it calculates the adjoint for the provided 4×4 matrix. When both the determinant and the adjoint are in place, the calculation of the inverse matrix is possible, and the code for this process can be executed, as coded in Fig. D.15.

The subroutines to define the control gains and the inverse matrix are only executed at the initialization stage. Afterward, a loop is executed for resolving the control terms each time a reference vector for the vehicle is received by the main board. To that end, the latest aircraft states

and the current value for their references are required. The user must generate these references, hence a reference signals generation routine is created. It takes the execution time steps as a parameter to be used in the pattern calculation to be tracked. The code in charge of this task is described in Fig. D.16 for trajectory tracking control. One important thing to notice is that the code does not get the target values instantly; in contrast, it generates integrable and derivable signals as the code needs time-continuous functions for its execution. Otherwise, the calculations present undetermined results, which are undesired.

During the error calculation for the control terms computing, the reference values and their first and second derivatives are used. For simplifying the access to them, they are provided in an ordered index accessible fashion which can be addressed with the state number minus one, due to C++ memory pointers math. For generating that format and providing ϕ, θ, ψ, z, x and y references and their derivatives, the ***GenerateXr*** function is defined in Fig. D.17, where the code covers the trajectory tracking case of the implementation.

Section 3.1 describes the essential need for calculating the errors and their derivatives between the reference signals and the current state variables. Based on those equations, ***UAVErrorCalculation*** is developed in Fig. D.18, and considering that ***GenerateXr*** partially provides the reference values, this function completes the 12 states array by defining $x_{2r}, x_{4r}, x_{6r}, x_{8r}, x_{10r}$ and x_{12r} that are necessary for the calculation of the errors $z_{2D}, z_{4D}, z_{6D}, z_{8D}, z_{10D}$ and z_{12D}, respectively.

The control algorithm requires an extra variable from the already listed. The angular speed difference generated by the rotors' configuration Ω is used in the control equations. Depending on the spinning direction (clockwise or counterclockwise) and rotor position, the effect that the rotor speed of an actuator takes in the formula, as defined in Table 1.1. The code for the implementation is the block of Fig. D.19. This shows the examples captured for $+$ and $\times$ quad-rotor configurations.

Finally, the control algorithm can be defined using the previous code subroutines. Fig. D.21 shows the example code for trajectory tracking flight mode using a backstepping scheme with linear control.

5.2.2.1 Integral and Derivative Routines

The implementation of the overall control scheme requires the calculation of the integrals and derivatives of certain variables. To cover that part, the implementation of two subroutines are

described in this section.

For the integral subroutine, different solutions are available where the most common options are the rectangular approximation, trapezoidal approximation, and the Simpson's method. The first one takes the sample period as a parameter to calculate the area below the curve using a rectangle between the sample $n-1$ and sample n as follows

$$i_n = i_{n-1} + Tx_{n-1} \tag{5-17}$$

where i_n is the current integral result, i_{n-1} is the previous result, T is the period between samples, and x_{n-1} is the previous sample value. The trapezoidal approximation of the integral, computes the area under the curve by using trapezoid shapes between samples, with the following formula

$$i_n = i_{n-1} + \left(\frac{T}{2}\right)(x - x_{n-1}) \tag{5-18}$$

where x is the last sample value. On the other hand, the Simpson's rule for integral approximation is slightly different from the previous two since it uses three samples to calculate the area below the curve using a second-order polynomial, as follows

$$i_n = i_{n-2} + \left(\frac{T}{3}\right)(x + 4x_{n-1} + x_{n-2}) \tag{5-19}$$

where i_{n-2} and x_{n-2} are the result and the sampled values from two steps before, correspondingly [Lyons-11].

When generating the source code for the Simulink integrator block, the inspection of the code indicates that the approximation used is the rectangular. Hence, in order to fit the simulation results, that approximation is used as well in the embedded implementation. The code for that block is shown in Fig. D.3, which represents a single integral.

It is important to mention that for every single integrator being used in the simulation, a different integrator routine is defined since the code needs to track the old result and previous value in order to calculate the integral, and that is being done by using static variables within the function.

The Levant differentiator as the first option to compute the necessary derivatives for the control algorithms. Its implementation requires integrals and sign functions and, as the integrators are already developed, the only thing remaining is the sign function routine, which is defined based on the following equation and its corresponding code is depicted in Fig. D.4.

$$\text{sign}\,(x) = \begin{cases} 1, if\ x > 0 \\ 0, if\ x = 0 \\ -1, if\ x < 0 \end{cases} \tag{5-20}$$

One complication that was faced during the implementation of Levant's differentiator is

that Simulink block uses a recursive term in the feedback loop of the integrators. For that reason, the embedded routine initializes the old values to 0 in the integrators. After the first call, the function updates the old value parameter in the integrator with the new result for later execution, to replicate the recursive mechanism of the loop. The code example for the integrators in Levant's differentiator of the X array is in Fig. D.5, while the complete differentiator code is in Fig. D.6.

During the code execution, it was noticed that the second derivatives required in section 2 expose some chattering when using the third-order Levant's derivative, which makes the control unstable. This is due to the chattering multiplication produced from the first derivative that migrated to the second derivative. In the cases where second or higher-order derivatives are required, the solution to the problem is to implement and compare a first-difference and central-difference approximation techniques for the discrete derivative. The first-difference differentiator is the process of computing the difference between successive x_n samples

$$y_n = \left(\frac{1}{T}\right)(x_n - x_{n-1}) \tag{5-21}$$

where y_n is the differentiator's output, T is the sampling period, x_n is the last sample and x_{n-1} is the previous sample. Fig. 5.9 shows the first and second derivative using the first-difference algorithm.

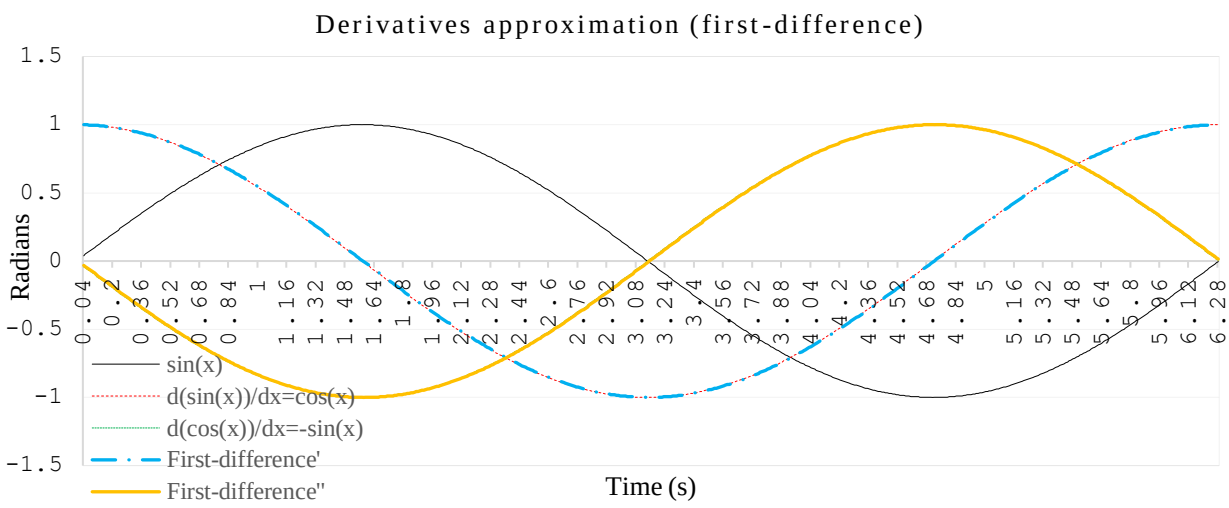


Fig. 5.9 First and second derivative approximations for a sinusoidal signal using the first-difference algorithm.

On the other hand, the central-difference differentiator is based on computing the average difference between alternate pairs of x_n samples [Lyons-11]

$$y_n = \frac{x_n + x_{n-2}}{2T}. \tag{5-22}$$

There are numerous approximations considering 5, 7, 9 or more data points, but during the data analysis of them, it was observed that the resulting derivative shifts the signal to the right, by the number of elements considered for the calculus times the sampling period. For the mentioned reason, the derivative is not realistic for the control purposes, thus the first-difference approximation is chosen for second or higher-order derivatives, Levant's are only used when single non-consecutive derivatives are required. Fig. 5.10 shows the first and second derivative approximations for a sinusoidal signal using the central-difference algorithm while Fig. 5.11 shows the Levant's approximations.

It is hard to notice the error between the first-difference algorithm and the central-difference method, therefore, Table 5.1 is provided to show the RMSE analysis between the algorithms for the second derivatives, where the first-difference method exposes the minimum error. The code for the derivatives calculation routine is displayed in Fig. D.7.

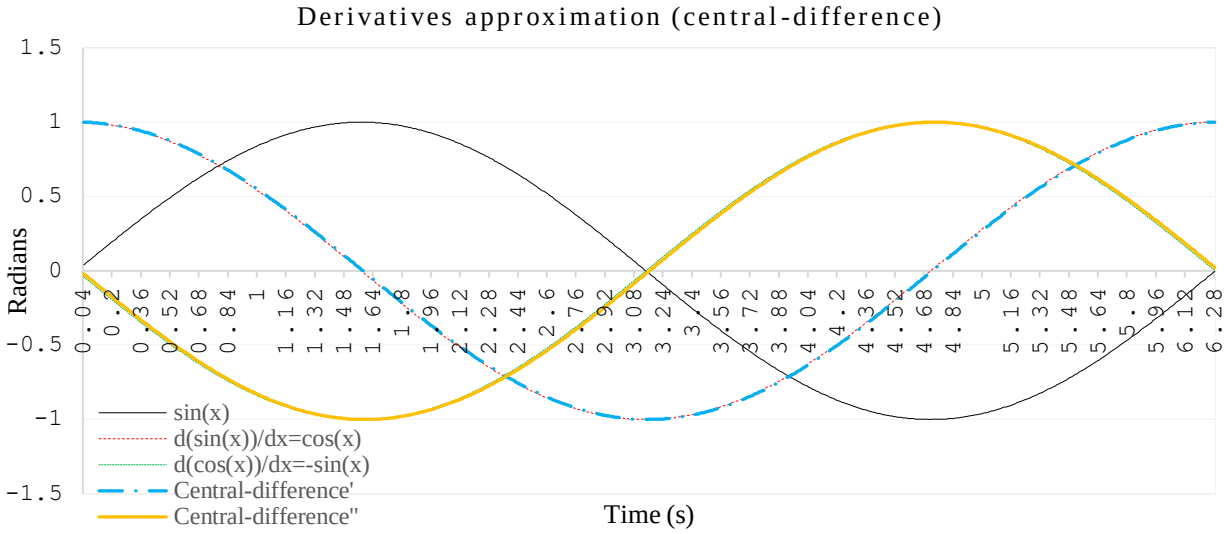


Fig. 5.10 First and second derivative approximations for a sinusoidal signal using the central-difference algorithm.

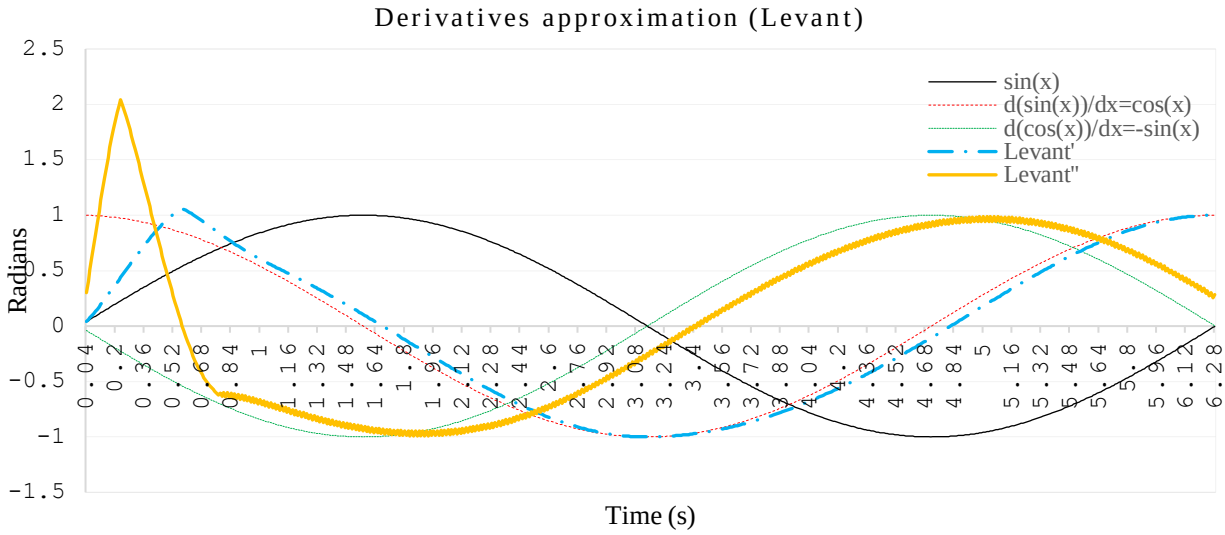


Fig. 5.11 First and second derivative approximations for a sinusoidal signal using Levant's algorithm.

TABLE 5.1. RMSE COMPARISON BETWEEN THE FIRST-DIFFERENCE, CENTRAL-DIFFERENCE AND LEVANT'S DIFFERENTIATOR FOR THE SECOND DERIVATIVE OF A SINUSOIDAL SIGNAL.

Calculus	First-difference	Central-difference	Levant
$\Sigma(e_t^2)$	0.031083657	0.124328521	139.0915905
$\Sigma\left(\frac{e_t^2}{n}\right)$	0.00004.97339	0.000198926	0.222546545
$\sqrt{\Sigma\left(\frac{e_t^2}{n}\right)}$	0.007052223	0.0141041	0.471748391

5.2.2.2 Motor Angular Speed Reference Routine

The control signals $U = [U_1, U_2, U_3, U_4]^T$ must be used to determine the references angular velocities ω_{1r}, ω_{2r}, ω_{3r} and ω_{4r} for the actuators. For this particular calculation, a block of code is defined in Fig. D.20, which uses U obtained from the code in Fig. D.21 and the inverse matrix M^{-1} defined in Fig. D.15. Based in (1-6), the speed is the square root of the multiplication of the inverse matrix by the array U. To not cause imaginary numbers calculations or change of spinning direction of the actuators, due to mechanical restrictions, a safety check is performed within the routine to analyze if the number to process with the square root is positive.

Additionally, the motor control block that is described in the next subsection requires the derivative of ω_{1r}, ω_{2r}, ω_{3r} and ω_{4r}. For that purpose, the output generated by ***CalcWr*** is derived using the Levant's derivative block in Fig. D.6.

5.2.2.3 Inner Loop Backstepping Controller Routine

The controller from section 5.2.2 provides the input parameters to be used for generating the motor speed references, nevertheless, it does not consider the motor dynamics of the BLDC motors used in the UAV. For that reason, an extra controller for tracking the reference velocity is required. In section 3.3 a proposal of a controller is made, and the same implementation is developed here for the embedded execution. Fig. D.22 shows the code that performs this task, which is based on a loop of the number of motors in the configuration to calculate each control for the specific BLDC rotors in the UAV frame.

5.2.2.4 Units Conversion Routines for the Controllers

The current consumption and speed of the motors are being captured in a raw format by the daughter-board. When calculating the controller, amperes, and revolutions per minute (RPM) units are required since these are the formats used in the simulation, by the multi-rotor mathematical model, and by the control algorithm. For that reason, two conversion routines are defined in the Pixhawk board code. In section 5.1.1.1, the ADC to current formula is defined and, by using that reference, the function from converting the ADC reading into milliamps is defined in Fig. D.1. In Fig. F.10 the timer count increase of 0.5 microseconds is defined. Since embedded system implementations do not normally support floating-point processing, a factor of 1×10^7 is used to convert the 0.5 μs into 5. The same multiplier is used to convert seconds into 0.1 μs units, in order to do the calculation properly. The routine for RPM conversion is defined in Fig. D.2.

5.3. Conclusions

The theorical and experimental implementation of a control scheme for any type of system require different technical capabilities. Usually, during the theoretical phase, only simulations are carried out and, consequently, a lot of environmental factors are discarded as highly nonlinear dynamics, external disturbances, and noise. For the proposal of this work, a two-loop control scheme is designed, and its implementation has required two electronic board as well: the main and the daughter-boards. In addition, current and speed sensing for brushless motors are not straightforward tasks since the sensors used generate fluctuating output. In addition to the basic

ADC and timer reads, a filtering process needs to be used for having an accurate system input. Therefore, the mean calculation for the ADC sampling process is performed when analyzing the digital conversions, along with the EWMA filter computation for the speed data. This data processing becomes essential for the control algorithm in order to reduce the chattering and high-frequency elements that can inject errors in the computation of the control signals. By avoiding these disturbances, the motors only handle governance control signals of the frequency and magnitude they can mechanically achieve.

6. Real-Time Experimental Results

Once the hardware and software architecture has been developed and functionally validated, the next stage is to develop real-time experiments to assess the performance of the control scheme and its embedded implementation from an integral point of view. The vehicle's parameters used during the experiments are depicted in Table 2.3 and Fig. 2.8, which correspond to the QAV250 UAV model.

To this end, first, experiments for the reference tracking of a single output are carried out. As a stabilize flight mode is selected, the outputs of the systems are the altitude (z), and the three Euler angles (θ, ϕ, ψ) corresponding to the attitude of the vehicle. Therefore, four different experiments for this stage were developed and reported. On the other hand, a test-bench is developed to mount the vehicle during the single output reference tracking experiments. This allows to finely tune the controller gains in a safe manner.

Finally, the performance of the overall control scheme for the reference tracking of the four outputs simultaneously is validated by means of free flight real time experiments. In this case, as the stabilize flight mode is set, the roll and pitch angles indirectly control the x and y positions of the aircraft. This device should be able to navigate in non-controlled environments, deal with the static error or handle variations such as changes in mass, payloads or inertia parameters. In the effort to test the robustness of the algorithms, disturbances are applied to the system to observe the functionality and tracking ability during flight.

6.1. Test-Bench Experiments

Previous to the free flight experiment, single output reference tracking experiments were carried out by using a test-bench. These tests structures allow the user to validate the controllers separately and to skip some parameters from the equation, as mass for example, but with a cost-benefit of changing the center of rotation in the aircraft to the point of joint with the mechanism. Fig. 6.1 shows the test-bench for the roll (ϕ) and pitch (θ) reference tracking experiments. It consists of a fixed metal tube that trespasses longitudinally the structure of the aircraft and it is

coupled to the vehicle by bearings which let the vehicle rotate around the fixed tube. Depending on the yaw position of the vehicle with respect the fixed tube, the configuration permits to test the pitch and roll rotation of the vehicle.

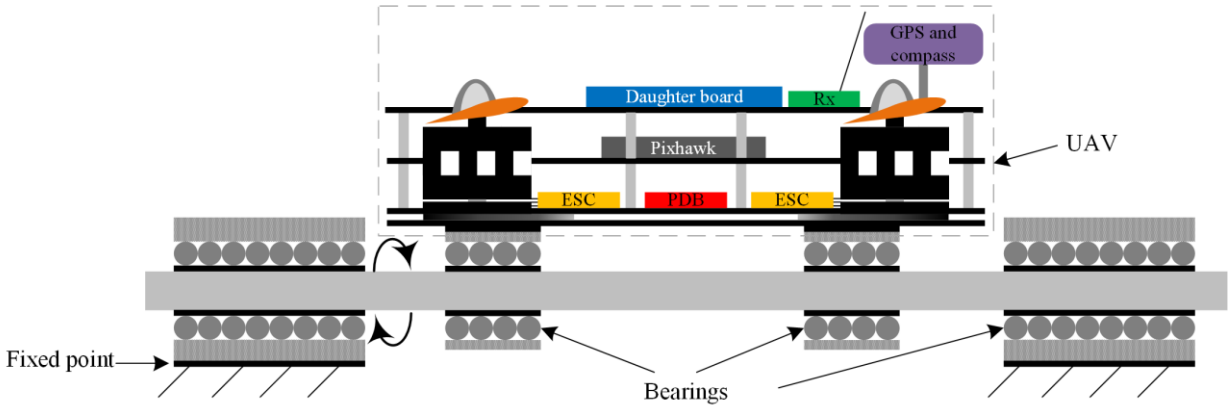


Fig. 6.1 Roll and pitch reference tracking control test-bench.

On the other hand, Fig. 6.2 depicts the test-bench for the yaw (ψ) and altitude (z) experiments. It is composed of an external tubular fixed base and an internal tubular base attached to the bottom part of the vehicle. This allows the vehicle to translate vertically and to rotate longitudinally around the fixed base.

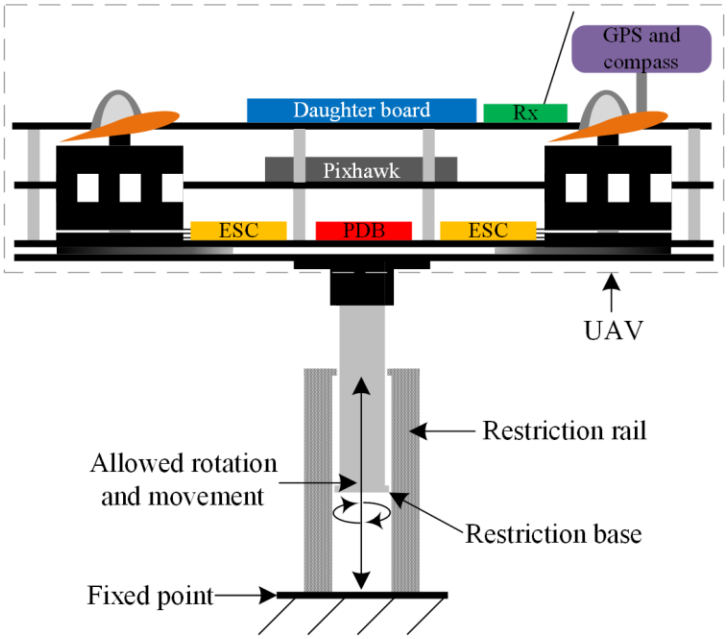


Fig. 6.2 Altitude and yaw angle reference tracking control test-bench.

6.1.1 Outer Control Loop Single Reference Tracking

The real time experiments were initially carried out selecting a single output of the vehicle on a test-bench designed to this end. The first experiment was developed by defining the output as the roll angle ϕ and considering the reference as $\phi_r = 0.4\cos(2\pi t)$ rad. This reference is determined by taking into consideration the mechanical limitations of the test-bench and the multi-rotor. Moreover, the reference includes a change in the sign of the angular velocity $\dot{\phi}$ which demands a higher performance from the actuators and the inner loop controller. The results are depicted in Fig. 6.3 considering the following control gains

$$k_1 = 20.53, k_2 = 20.53. \tag{6-1}$$

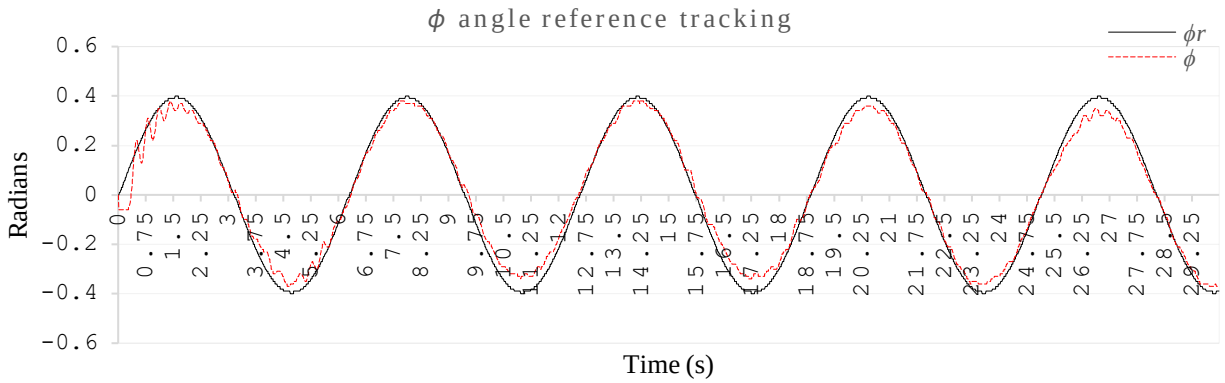


Fig. 6.3 Test-bench experiment results for the reference tracking of the vehicle's roll angle ϕ.

It can be noted that the reference tracking is developed successfully. As expected, during the sign change of the angular velocity $\dot{\phi}$ the actuators performance is decreased as their behavior is stressed. However, the overall experiment for this individual output is satisfactory.

Then, the same reference vector $\theta_r = 0.4\cos(2\pi t)$ rad is defined for the pitch angle θ of the vehicle. The result of the experiment is shown in Fig. 6.4, considering the following control gains

$$k_3 = 40.92, k_4 = 7.83, \tag{6-2}$$

and demonstrating the convergence of the error variable to zero.

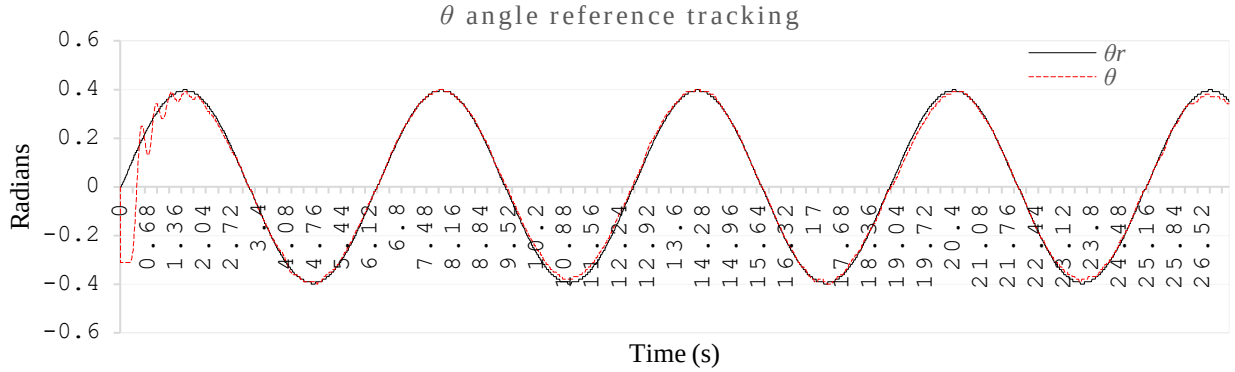


Fig. 6.4 Test-bench experiment results for the reference tracking of the vehicle's pitch angle θ.

In addition, the performance of the reference tracking for the pitch angle is better than the one obtained for roll angle as it is not affected by the sign change of $\dot{\theta}$.

For the yaw angle ψ, the reference vector is defined as $\psi_r = 0.8\cos(2\pi t)\,\text{rad}$ and the results of the experiments are depicted in Fig. 6.5 with the control gains defined as

$$k_5 = 15, k_6 = 21.5. \tag{6-3}$$

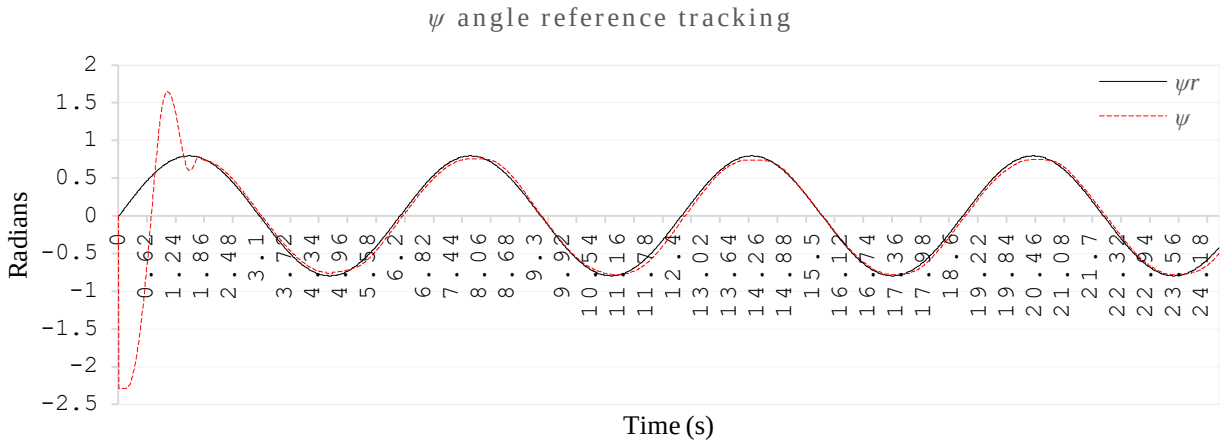


Fig. 6.5 Test-bench experiment results for the reference tracking of the vehicle's yaw angle ψ.

The figure shows that the control objective is fulfilled but the transient part of the response is characterized by a considerable large overdamping. This can be attributed to the inertia generated by the initial error value and to the well-known inability of the linear control terms, utilized during the backstepping control algorithm, to compensate this phenomenon.

Finally, the altitude z reference tracking experiment was carried out considering the

reference $z_r = 0.28 + 0.03\cos(2\pi t)$ m, and considering $\phi_r = \theta_r = \psi_r = 0$. The resulting control gains are given by

$$k_7 = 9.55, k_8 = 9.6, \tag{6-4}$$

and the experimental result is shown in Fig. 6.6.

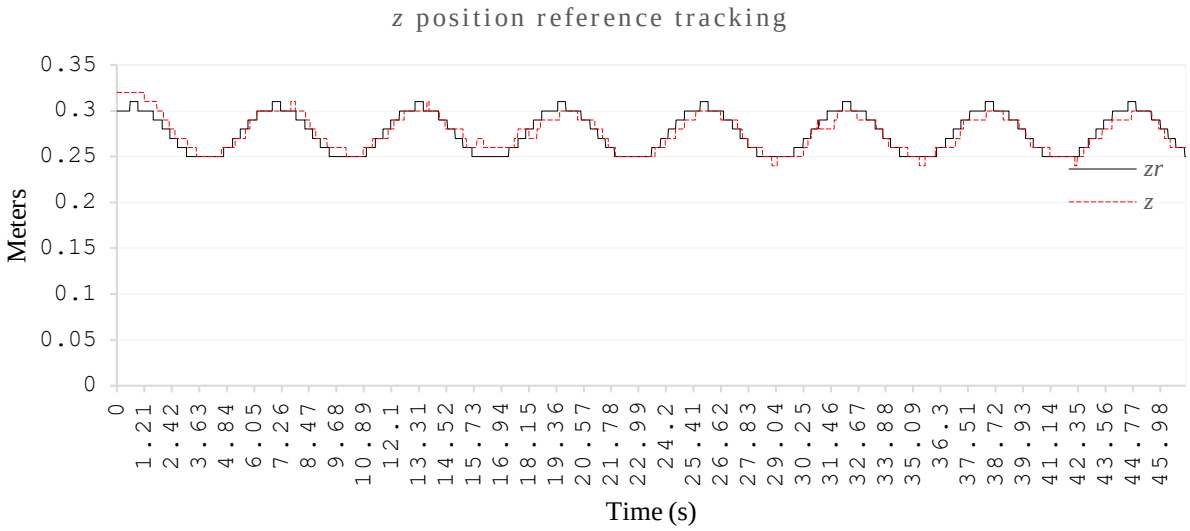


Fig. 6.6 Test-bench experiment results for the reference tracking of the vehicle's altitude z.

The results of the four single output reference tracking can be appreciated in [Mayorga-Macías-20] where the real time experiments are presented. The use of the test-benches for the reference tracking of a single output permitted to demonstrate the performance of the controller and to heuristically tune the controller gains by means of the repetition of the experiment until the desired performance was obtained. This process allowed to tune the inner control loop gains as well, which resulted with the value $k_{iM} = 305$ for $i = 1,2,3,4$.

Evidently, there is a price to pay for the use of the test-benches during the controller gains tuning process. It is related to the undesirable and unmodelled dynamics that the test benches add to the closed loop system which are originated by the inertia of the benches' components that were coupled to the vehicle, and the friction of their degrees of freedom. Nonetheless, the validation of the performance of the overall controller without these mechanical limitations is presented in the following section of this work.

6.2. Free Flight Experiments

Once the experiments with the test benches have been done, resulting in the tuning of all the controller gains, the experiments with the aircraft in free flight are ready to be carried out.

For these experiments, the reference for the yaw angle ψ_r of the vehicle was defined as zero, and the references for pitch θ_r and roll ϕ_r angles were defined to generate a square path for the coordinates x and y of the vehicle, by means of the stabilize flight mode. To this end, two sinusoidal components for the references θ_r and ϕ_r were generated: the initial and larger component with 0.18 rad of amplitude and a terminal and smaller component with amplitude of 0.04 rad which is used to stop cancel inertia of the vehicle generated by its previous motion. Finally, the altitude reference z_r was designed to obtain a smooth transition from zero altitude to 0.7 m and vice versa. It aims to develop a safer take-off and landing of the vehicle during the experiments.

The results for the experiment are depicted in Fig. 6.7, Fig. 6.8, Fig. 6.9 and Fig. 6.10 where the tracking response for ϕ, θ, ψ and z are shown, respectively.

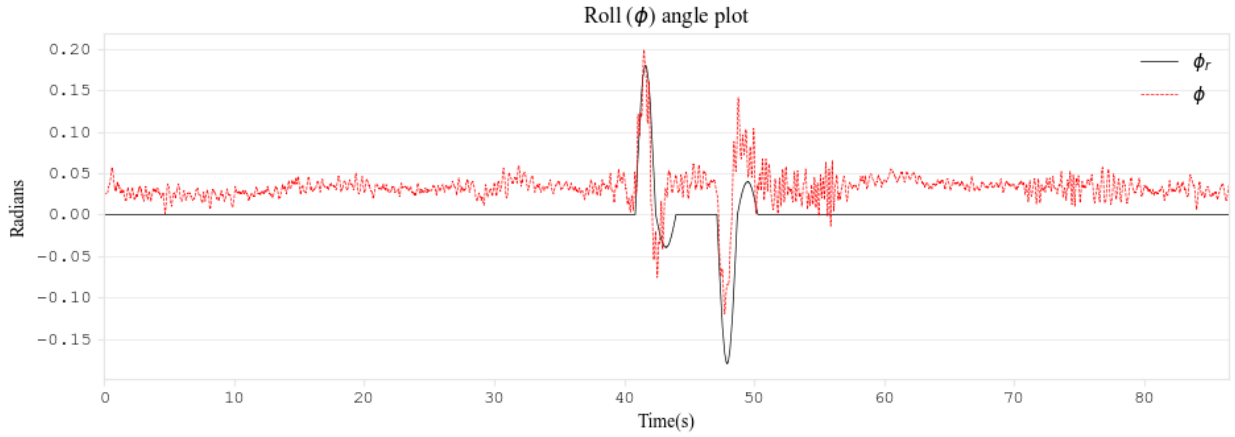


Fig. 6.7 Free flight experiment results for the reference tracking of the vehicle's roll angle ϕ.

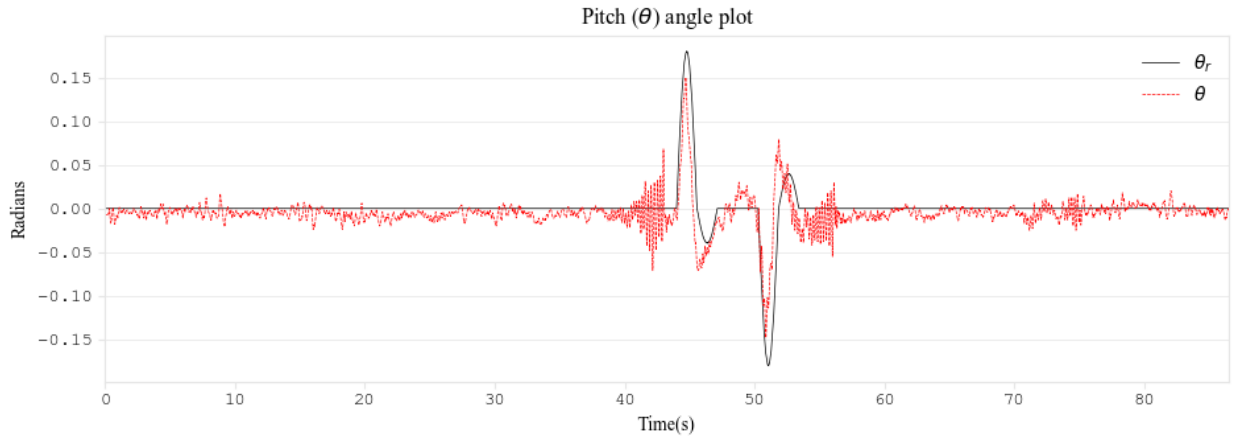


Fig. 6.8 Free flight experiment results for the reference tracking of the vehicle's pitch angle θ.

Fig. 6.9 Free flight experiment results for the reference tracking of the vehicle's yaw angle ψ.

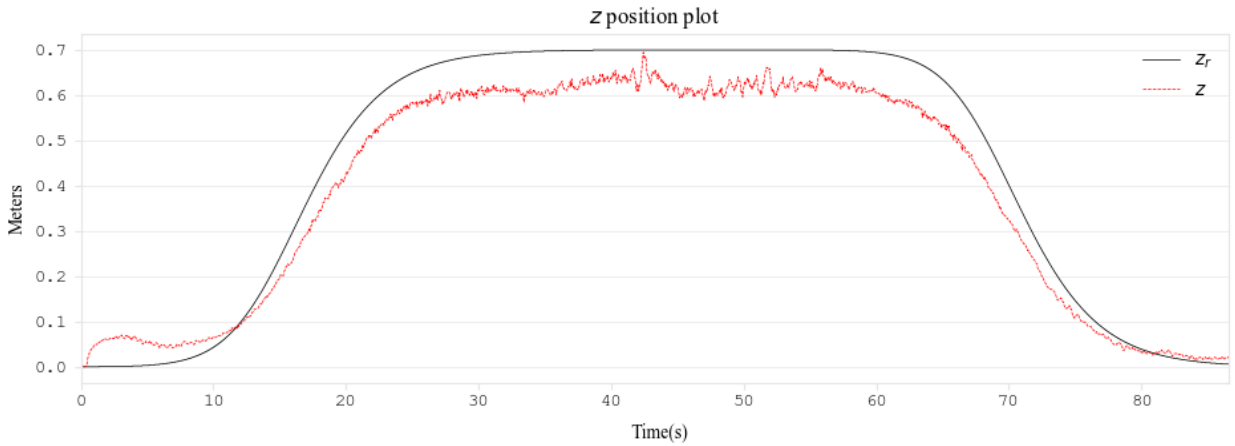


Fig. 6.10 Free flight experiment results for the reference tracking of the vehicle's altitude z.

It can be noted that the best performance is obtained for the reference tracking of the pitch angle θ_r, as the performance for the tracking of other three references show a significant steady state error. Moreover, the worst performance was obtained for the tracking of the yaw angle reference ψ_r as it shows the larger tracking error amplitude.

This can also be concluded by analyzing Table 6.1 where the root-mean-square error for each of the tracked references of the experiment are displayed. Nonetheless, the control objective was achieved, and its results can be appreciated in detail in [González-Jiménez-20], where a video of the free flight real time experiment is shown.

TABLE 6.1. RMSE CALCULATION FOR THE REFERENCE TRACKING OF THE VEHICLE IN FREE FLIGHT

Calculus	ϕ	θ	ψ	z
$\Sigma(e_t^2)$	17.8588 rad^2	5.1136 rad^2	5.4792 rad^2	0.3087 m^2
$\Sigma\left(\frac{e_t^2}{n}\right)$	0.002443733 rad^2	0.000699726 rad^2	0.000749754 rad^2	0.0000655414 m^2
$\sqrt{\Sigma\left(\frac{e_t^2}{n}\right)}$	0.049434127 rad	0.026452341 rad	0.027381631 rad	0.008095764 m

It is important to mention that, in contrast with test-bench experiments, these experimental results were affected by more external disturbances and unmodelled dynamics. For instance, the ArduPilot code provides centimeter granularity for the altitude sensing independently of the sensor capabilities, which causes size considerable error in the derivatives due to the discretization method. Hence, instead of using the centimeter granularity approximation, the code was modified to accumulate the millimeters readings from VL53L0XV2, and then send the value to the controller. In addition, noise was present in the altitude measurement as well, and an EWMA filter was applied. On the other hand, the payload of the aircraft is defined as 0.85 kg, but the experiment was performed with additional 50 gr. of mass attached to the aircraft, which composes a parametric variation to the system. Also, the moments of inertia use the mass parameter to calculate its value, and since the factor was changed, I_x, I_y, and I_z compose not modeled dynamics. Moreover, three-phase brushless motors are used in the vehicle, but their models were approximated considering single-phase DC models. This adds another perturbation to the controller that needs to be addressed in future works.

6.3. Conclusions

Code implementation and its debugging perform an essential role when implementing control algorithm for a real time experiment with a UAV. These tasks are not designed during the controller design stage, so a part of their development is carried out during the experiments and the designer must troubleshoot as the issues are detected. Usually, these issues are generated due to unconsidered subsystems in the dynamical model of the vehicle, noise in the instrumentation of the vehicle, external disturbances, and parametric variations.

Despite all these factors, this chapter showed successful results during the real time experiments for the embedded implementation of the overall control scheme designed. The selected flight mode was Stabilize, which means that the controller receives reference signals for the attitude (roll, pitch and yaw) and the altitude if the aircraft. Therefore, the longitudinal and lateral coordinates of the UAV are indirectly controlled by the pitch and roll angle, respectively. This principle was used to generate the reference for the experiments to generate a rectangular path for the position of the vehicle.

It is worth to note, that this work considers the backstepping control technique to synthetize both control loops of the overall scheme, but there are plenty other algorithms to be investigated and implemented with better features and characteristics related to robustness, smooth convergence, energy optimality or adaptability.

General Conclusions

In this work, the design and implementation of a control scheme for the reference tracking of a multi-rotor in Stabilization flight mode was presented. In this flight mode, the controller receives four reference signals: three for the Euler angles that define the orientation of the aircraft and one more for its altitude.

As a first step in the design of the proposed control scheme, the differential equations that define the dynamics of the vehicle and its actuators were obtained. From these equations, a state space model was obtained with 12 state variables corresponding to the linear and angular positions and velocities of the UAV. The actuators of this type of vehicle are brushless electric motors (BLDC) whose power interface is known as ESC, from Electronic Speed Controller. To avoid the complexity of the equations of this type of motor and the characterization of its parameters, transfer functions (TF) of the actuators were obtained experimentally considering the duty cycle of the PWM signal that is introduced to the ESC of the motor and as outputs the angular speed of the motor and its current.

Once a dynamic model of the complete system and its actuators had been obtained, a control scheme with two feedback loops was proposed: an outer loop that controls the dynamics of the vehicle, and an inner loop that controls the dynamics of the actuators. Based on this scheme, three control algorithms were designed for the external loop using three different techniques: backstepping, sliding modes, and PID. These three controllers were designed for two different flight modes: Stabilization and Trajectory. In addition, the backstepping technique was also used to design the inner loop controller of the global scheme. Using the dynamic models of the vehicle and actuators, and the controllers for the external and internal loops, simulation results were obtained that validated the effectiveness of the proposed algorithms.

Of the two flight modes considered in simulation, Stabilization was chosen to carry out the embedded implementation and the experiments in real time. This is because it was considered to carry out the experiments indoors, and there was no sensor available that would allow us to feedback the position of the vehicle accurately and precisely. Additionally, the backstepping based controller were chosen for both feedback loops due to the low robustness of the PID controller and the high-frequency components that slide modes induce in system response and the generated

control signals.

After the simulation stage, the algorithms were implemented in a system embedded in the aircraft. The implementation required, in addition to coding the algorithm, designing an angular velocity sensor for the actuator axles, and conditioning the signals measured by the current sensors. To make the implementation of these stages more efficient and have more input / output ports in the embedded system, a daughter embedded board was integrated to the platform that was in charge of the acquisition and instrumentation stages of said sensors, and to maintain constant communication with the main embedded system. The control algorithms for both feedback loops were encoded on the main board.

Finally, the experiments were carried out in real time with the embedded implementation of the designed global control scheme. For safety and gain tuning purposes, the experiments were first conducted using a test bench that limited the degrees of freedom of the vehicle. This made it possible to validate the control algorithm for tracking the references individually. Once a good performance was obtained under this modality, the obtained gains were used for the implementation of the proposed control scheme with the vehicle in free flight. These experiments were successful as all four references were tracked simultaneously.

On the other hand, implementing a control scheme for the vehicle that considers the sensing of the current consumed by the actuators allows the detection of faults in the motors and the reconfiguration of the controller to increase its robustness during the occurrence of these events [Chen-14], [Merheb], [Santos], [Qian-16].

Moreover, the system can easily implement different control algorithms that can improve the performance of the closed-loop system. The only requirement, from a code development point of view, is the modification of the control routines implemented on the main embedded system.

Assuming the proper control of the aircraft, other research fields can be developed to give more autonomy to the device. One approach that can be investigated is the use of clustering for data compression in images taken from the aircraft to classify them in three different categories as left, center, and right with the purpose of obtaining the yaw angle of the vehicle with respect to a desired pathway. It will allow to the controller to orient the aircraft properly and to continue the flight in the desired route. This area of research requires the implementation of a neural network method and image sets for training the system, to obtain the array of weights of the controller that drives the UAV [Giusti-15].

Additionally, some improvements were identified from the embedded implementation point of view. The variables and calculations that require floating-point support use *double* (8 bytes) data type; nevertheless, the level of precision required for the computation is not at that level, therefore the float (4 bytes) data type could be used, decreasing the data memory required by the storing device.

One of the limitations faced regarding the execution frequency for the controllers in this dissertation, was the developer restrictions from the ArduCopter scheduler as 10ms is the fastest pre-defined task that does not interrupt the other routines running. This limits the update frequency for the control outputs and in order to accelerate the processing, a new software architecture needs to be selected from the pre-existing open source solutions that provide alternatives to the developer, otherwise, implement a new software and hardware architecture.

Conclusiones Generales

En este trabajo se presentó el diseño e implementación de un esquema de control para el seguimiento de referencia de un multi-rotor en modo de vuelo Estabilización. En este modo de vuelo, el controlador recibe cuatro señales de referencia: tres para los ángulos de Euler que definen la orientación de la aeronave y uno más para su altitud.

Como primer paso para el diseño del esquema de control propuesto, se obtuvieron las ecuaciones diferenciales que definen la dinámica del vehículo y de sus actuadores. De estas ecuaciones, se obtuvo un modelo en espacio de estados con 12 variables de estado correspondientes a las posiciones y velocidades lineales y angulares del UAV. Los actuadores de este tipo de vehículos son motores eléctricos sin escobillas (BLDC) cuya interfaz de potencia se conoce como ESC, del inglés *Electronic Speed Controller*. Para evitar la complejidad de las ecuaciones de este tipo de motor y la caracterización de sus parámetros, se obtuvieron experimentalmente funciones de transferencia (FDT) de los actuadores considerando el ciclo de trabajo de la señal PWM que se introduce al ESC del motor y como salidas la velocidad angular del motor y su corriente.

Una vez obtenido un modelo dinámico del sistema completo y sus actuadores, se planteó un esquema de control con dos lazos de retroalimentación: un lazo externo que controla la dinámica del vehículo, y un lazo interno que controla la dinámica de los actuadores. Basado en este esquema, se diseñaron tres algoritmos de control para el lazo externo usando tres diferentes técnicas: *backstepping*, modos deslizantes y PID. Estos tres controladores se diseñaron en dos diferentes modos de vuelo: Estabilización y Trayectoria. Además, se utilizó la técnica de *backstepping* también para diseñar el controlador del lazo interno del esquema global. Usando los modelos dinámicos del vehículo y los actuadores, y los controladores para los lazos externo e interno, se obtuvieron resultados de simulación que validaron la efectividad de los algoritmos propuestos.

De los dos modos de vuelos considerados en simulación, se eligió el de Estabilización para realizar la implementación embebida y los experimentos en tiempo real. Esto debido a que se planteó realizar dichos experimentos en interiores y no se contaba con un sensor que nos permitiera retroalimentar de forma exacta y precisa la posición del vehículo durante los experimentos.

Además, se eligieron los controladores basados en *backstepping* para ambos lazos de retroalimentación debido a la baja robustez del controlador PID y a las componentes de alta frecuencia que los modos deslizantes inducen en la respuesta del sistema y las señales de control generadas.

Después de la etapa de simulación, se realizó la implementación de los algoritmos en un sistema embebido en la aeronave. La implementación requirió, además de la codificación del algoritmo, el diseñar un sensor de velocidad angular para los ejes de los actuadores y el acondicionar las señales medidas por los sensores de corriente. Para hacer más eficiente la implementación de estas etapas y contar con más puertos entrada/salida en el sistema embebido, se integró una tarjeta de procesamiento auxiliar que se encargó de las etapas de adquisición e instrumentación de dichos sensores, y mantener constante comunicación con la tarjeta de procesamiento principal. En la tarjeta principal se codificaron los algoritmos de control de ambos lazos de retroalimentación.

Finalmente, se realizaron los experimentos en tiempo real con la implementación embebida del esquema global de control diseñado. Por cuestiones de seguridad y sintonización de ganancias, primero se realizaron los experimentos mediante una cama de pruebas que limitaba los grados de libertad solo a algunas de las variables del vehículo. Esto permitía validar el algoritmo de control para el seguimiento de las referencias de forma individual. Una vez obtenido un buen desempeño bajo esta modalidad, se usaron las ganancias obtenidas para la implementación del esquema propuesto de control con el vehículo en vuelo libre. Estos experimentos resultaron satisfactorios al lograrse el seguimiento de las cuatro referencias simultáneamente.

Por otro lado, el instrumentar un esquema de control para el vehículo que considera el sensado de la corriente consumida por los actuadores permite la detección de fallas en los motores y la reconfiguración del controlador para incrementar su robustez ante estos eventos [Chen-14], [Merheb], [Santos], [Qian-16].

Además, el sistema es capaz de implementar de forma sencilla diferentes algoritmos de control que pueden mejorar el desempeño del sistema en lazo cerrado. El único requerimiento, desde el punto de vista de desarrollo de código, es la modificación de las rutinas de control implementadas en la tarjeta de procesamiento principal.

Con el control adecuado de la aeronave, otras áreas pueden ser involucradas en el desarrollo para dar autonomía al dispositivo. Una de las aproximaciones que pueden ser investigadas es la

del uso de *clustering* para compresión de datos en imágenes obtenidas desde la nave, y después clasificarlas en tres diferentes categorías como izquierda, centro, y derecha con el propósito de saber la orientación en el ángulo de *yaw* en el sistema sobre un camino o vereda, para finalmente orientar el dispositivo en su posición central y entonces continuar el vuelo en la ruta deseada. Esta área de investigación requiere de la implementación de alguno de los métodos de redes neuronales, y conjuntos imágenes para entrenar el sistema y con ello obtener el arreglo de pesos para calcular las salidas de control para la orientación del VANT [Giusti-15].

Adicionalmente a lo ya mencionado, se identifican algunas mejoras a realizar en la implementación embebida. Las variables y cálculos que requieren soporte de punto flotante hacen uso de tipos de dato doble (8 bytes); sin embargo, la precisión necesaria para el cómputo no requiere de tal nivel precisión, por lo que el tipo de dato usado en los algoritmos podría ser cambiado a flotante (4 bytes), disminuyendo con ello la memoria de datos requerida.

Una de las limitantes con respecto a la frecuencia de ejecución de los controladores en este trabajo, fue la restricción de tareas definidas en ArduCopter para su uso por los desarrolladores, siendo de 10 ms la tarea predefinida más rápida que no interrumpe las tareas programadas, limitando a esa frecuencia la actualización de las salidas de control. Para acelerar el procesamiento, nueva arquitectura de software debe ser elegida, ya sea en soluciones existentes de código abierto que proporcionen alternativas al desarrollador, o en su caso, la implementación de una nueva arquitectura tanto de software como de hardware.

Appendix

A. LIST OF INTERNAL RESEARCH REPORTS

1) W. A. Mayorga-Macías, L. E. González-Jiménez, and L. Luque-Vega, "Mathematical model and flying modes for hexacopters by using backstepping," Internal Report *PhDEngScITESO-15-04-R*, ITESO, Tlaquepaque, Mexico, May 2015.

2) W. A. Mayorga-Macías, L. E. González-Jiménez, and L. Luque-Vega, "Mathematical model and flying modes for hexacopters by using backstepping and sliding mode control," Internal Report *PhDEngScITESO-15-07-R*, ITESO, Tlaquepaque, Mexico, Jul. 2015.

3) W. A. Mayorga-Macías, L. E. González-Jiménez, and L. Luque-Vega, "PID control for unmanned aerial vehicles and comparison with backstepping and sliding mode control," Internal Report *PhDEngScITESO-16-07-R*, ITESO, Tlaquepaque, Mexico, Jun. 2016.

4) W. A. Mayorga-Macías, L. E. González-Jiménez, and L. Luque-Vega, "Hexarotor control model and rotor fault tolerance," Internal Report *PhDEngScITESO-17-35-R*, ITESO, Tlaquepaque, Mexico, Nov. 2017.

5) W. A. Mayorga-Macías, M. A. Meza-Aguilar, and L. E. González-Jiménez, "DC motor mathematical model and control for multicopters," Internal Report *PhDEngScITESO-18-01-R*, ITESO, Tlaquepaque, Mexico, Mar. 2018.

6) W. A. Mayorga-Macías, M. A. Meza-Aguilar, and L. E. González-Jiménez, "DC current sensing for multicopter motor control," Internal Report *PhDEngScITESO-18-40-R*, ITESO, Tlaquepaque, Mexico, Dec. 2018.

7) W. A. Mayorga-Macías, M. A. Meza-Aguilar, and L. E. González-Jiménez, "Modeling of permanent magnet synchronous motors using current sensors," Internal Report *PhDEngScITESO-18-44-R*, ITESO, Tlaquepaque, Mexico, Dec. 2018.

8) W. A. Mayorga-Macías, M. A. Meza-Aguilar, and L. E. González-Jiménez, "Voltage-Speed transfer function for multi-rotor actuators and control code requirements," Internal Report *PhDEngScITESO-18-50-R*, ITESO, Tlaquepaque, Mexico, Dec. 2018.

9) W. A. Mayorga-Macías, M. A. Meza-Aguilar, and L. E. González-Jiménez, "Multirotor control algorithm implementation in an embedded system," Internal Report *PhDEngScITESO-18-55-R*, ITESO, Tlaquepaque, Mexico, Dec. 2018.

10) W. A. Mayorga-Macías, M. A. Meza-Aguilar, and L. E. González-Jiménez, "Embedded stabilization control for a multi-rotor," Internal Report *PhDEngScITESO-19-20-R*, ITESO, Tlaquepaque, Mexico, Nov. 2019.

11) W. A. Mayorga-Macías, M. A. Meza-Aguilar, and L. E. González-Jiménez, "Embedded implementation of a robust reference tracking controller for a multi-rotor," Internal Report *PhDEngScITESO-19-24-R*, ITESO, Tlaquepaque, Mexico, Dec. 2019.

B. LIST OF PUBLICATIONS

B.1. Book Chapters

1) G. Jouhet, L. E. González-Jiménez, M. A. Meza-Aguilar, **W. A. Mayorga-Macías**, and L. F. Luque-Vega, "Model-Based Fault Detection of Permanent Magnet Synchronous Motors of Drones Using Current Sensors", in New Trends in Robot Control, 1st ed., vol. 270, J. Ghommam, N. Derbel, and Q. Zhu, Ed. Singapore: Springer, 2020, pp. 301-318. (p-ISBN: 978-981-15-1818-8; e-ISBN: 978-981-15-1819-5; p-ISSN: 2198-4182; e-ISSN: 2198-4190; published: 2020; DOI: 10.1007/978-981-15-1819-5)

B.2. Journal Papers

1) **W. A. Mayorga-Macías**, L. E. González-Jiménez, M. A. Meza-Aguilar, and L. F. Luque-Vega, "Low-Cost Experimental Methodology for the Dynamic Model Approximation of Multirotor Actuators," *J. of Aerospace Engineering*, vol. 2020, pp. 1-9, Jul. 2020. (e-ISSN: 1687-5966; published online: 11 July 2020; DOI: 10.1155/2020/9263961)

2) **W. A. Mayorga-Macías**, L. E. González-Jiménez, M. A. Meza-Aguilar, and L. F. Luque-Vega, "Velocity Sensor for Real-Time Backstepping Control of a Multirotor," *Sensors*, vol. 2020, pp. 1-19, Jul. 2020. (e-ISSN: 1424-8220; published online: 29 July 2020; DOI: 10.3390/s20154229)

C. UAV SIMULATION SOURCE CODE

```
function [Yaw, Z, X, Y] = main(t)
  Yaw = 0; Zr  = 3; Xr  = 1; Yr  = 2; slope = 2.9;
  t1 = 4; t2 = 8; t3 = 12; t4 = 16; %Times
  %Positions
  Z = Zr/(1 + exp (-slope*(t-(5/slope))))^2;
  X = Xr/(1 + exp (-slope*(t-t1-(5/slope))))^2 -
      Xr/(1 + exp (-slope*(t-t3-(5/slope))))^2;
  Y = Yr/(1 + exp (-slope*(t-t2-(5/slope))))^2 -
      Yr/(1 + exp (-slope*(t-t4-(5/slope))))^2;
end
```

Fig. C.1 Yaw, x, y and z reference definition for simulation.

```
function U = main(m, g, l, Ix, Iy, Iz, Jr, k, omega, x, xrp, z, zp)
  %Parameters  1          2          3          4          5
  a  = [((Iy - Iz)/Ix) (-Jr/Ix) ((Iz - Ix)/Iy) (Jr/Iy) ((Ix - Iy)/Iz)];
  %       1      2      3
  b  = [(l/Ix) (l/Iy) (l/Iz)];

  %Z position control
  U1 = (m/(cos(x(1))*cos(x(3))))*(xrp(8) + k(7)*zp(7) + g + z(7) +
       k(8)*z(8));

  %Roll angle control
  U2 = (1/b(1))*(xrp(2) + k(1)*zp(1) - x(4)*x(6)*a(1) - x(4)*a(2)*omega +
       z(1) + k(2)*z(2));

  %Pitch angle control
  U3 = (1/b(2))*(xrp(4) + k(3)*zp(3) - x(2)*x(6)*a(3) - x(2)*a(4)*omega +
       z(3) + k(4)*z(4));

  %Yaw angle control
  U4 = (1/b(3))*(xrp(6) + k(5)*zp(5) - x(2)*x(4)*a(5) + z(5) + k(6)*z(6));
  U = [U1; U2; U3; U4];
end
```

Fig. C.2 UAV backstepping control equations for stabilize flight mode.

```
function [U,x1r,x3r] = main(m, g, l, Ix, Iy, Iz, Jr, k, omega, x, xrp, z, zp)
  %Parameters
  %           1          2          3          4          5
  a  = [((Iy - Iz)/Ix) (-Jr/Ix) ((Iz - Ix)/Iy) (Jr/Iy) ((Ix - Iy)/Iz)];
  %       1      2      3
  b  = [(l/Ix) (l/Iy) (l/Iz)];

  %Z position control
  U1 = (m/(cos(x(1))*cos(x(3))))*(xrp(8) + k(7)*zp(7) + g + z(7)*abs(z(8)) + k(8)*sign(z(8)));

  %Roll angle control
  U2 = (1/b(1))*(xrp(2) + k(1)*zp(1) - x(4)*x(6)*a(1) - x(4)*a(2)*omega + z(1)*abs(z(2)) +
       k(2)*sign(z(2)));

  %Pitch angle control
  U3 = (1/b(2))*(xrp(4) + k(3)*zp(3) - x(2)*x(6)*a(3) - x(2)*a(4)*omega + z(3)*abs(z(4)) +
       k(4)*sign(z(4)));

  %Yaw angle control
  U4 = (1/b(3))*(xrp(6) + k(5)*zp(5) - x(2)*x(4)*a(5) + z(5)*abs(z(6)) + k(6)*sign(z(6)));
  U = [U1; U2; U3; U4];
end
```

Fig. C.3 UAV backstepping sliding mode control equations for stabilize flight mode.

```
function dx  = main(x, M, W_rps, m, g, l, Ix, Iy, Iz, Jr, xr, omega)
  %            1          2          3          4          5
  a  = [((Iy - Iz)/Ix) (-Jr/Ix) ((Iz - Ix)/Iy) (Jr/Iy) ((Ix - Iy)/Iz)];
  %       1      2      3
  b  = [(l/Ix) (l/Iy) (l/Iz)];
  %Note: Use RPS instead of RPM
  U = M * power(W_rps, 2);
  %Variables
  ux = (cos(x(1))*sin(x(3))*cos(x(5)) + sin(x(1))*sin(x(5)));
  uy = (cos(x(1))*sin(x(3))*sin(x(5)) - sin(x(1))*cos(x(5)));
  %Roll(x)
  dx1  = x(2);
  dx2  = x(4)*x(6)*a(1) + x(4)*a(2)*omega + b(1)*U(2);
  %Pitch(y)
  dx3  = x(4);
  dx4  = x(2)*x(6)*a(3) + x(2)*a(4)*omega + b(2)*U(3);
  %Yaw(z)
  dx5  = x(6);
  dx6  = x(4)*x(2)*a(5) + b(3)*U(4);
  %Z
  dx7  = x(8);
  dx8  = -g + (cos(x(1))*cos(x(3)))*(1/m)*U(1);
  %X
  dx9  = x(10);
  dx10 = ux*(1/m)*U(1);
  %Y
  dx11 = x(12);
  dx12 = uy*(1/m)*U(1);
  dx = [dx1; dx2; dx3; dx4; dx5; dx6; dx7; dx8; dx9; dx10; dx11; dx12];
end
```

Fig. C.4 UAV mathematical model.

```
function [xrV, z, zp] = main(x, xr, k)
  xrV = zeros([length(xr), 1]);
  z   = zeros([length(xr), 1]);
  zp  = zeros([length(xr), 1]);
  %Roll angle control errors
  z(1)    = xr(1) - x(1);         zp(1)   = xr(2) - x(2);
  xrV(2)  = xr(2) + k(1)*z(1);    z(2)    = xrV(2) - x(2);
  %Pitch angle control errors
  z(3)    = xr(3) - x(3);         zp(3)   = xr(4) - x(4);
  xrV(4)  = xr(4) + k(3)*z(3);    z(4)    = xrV(4) - x(4);
  %Yaw angle control errors
  z(5)    = xr(5) - x(5);         zp(5)   = xr(6) - x(6);
  xrV(6)  = xr(6) + k(5)*z(5);    z(6)    = xrV(6) - x(6);
  %Z position control errors
  z(7)    = xr(7) - x(7);         zp(7)   = xr(8) - x(8);
  xrV(8)  = xr(8) + k(7)*z(7);    z(8)    = xrV(8) - x(8);
  %X position control errors
  z(9)    = xr(9) - x(9);         zp(9)   = xr(10) - x(10);
  xrV(10) = xr(10) + k(9)*z(9);   z(10)   = xrV(10) - x(10);
  %Y position control errors
  z(11)   = xr(11) - x(11);       zp(11)  = xr(12) - x(12);
  xrV(12) = xr(12) + k(11)*z(11); z(12)   = xrV(12) - x(12);
end
```

Fig. C.5 UAV states errors equations.

```
function [M, Minv, FConfig]= main(Ct,Cd,Rho,D,l,LSide,SSide,Clk,FThr,MTOff)
  % 3+  1-       ^ x QAV 250 configuration (Pixhawk)
  %  \^/         |
  %   o    y < - o
  %  / \
  % 2-  4+
  b = Ct*Rho*D^4;
  p = b*LSide/l; %sin(52.56)=opposite/hypotenuse=0.7941(Roll)
  q = b*SSide/l; %sin(37.44)=opposite/hypotenuse=0.6078(Pitch)
  d = Cd*Rho*D^4;
  %     M1 M2 M3 M4
  A = [ b  b  b  b];
  B = [ p -p -p  p];
  C = [ q -q  q -q];
  D = [ d  d -d -d];
  FConfig = 2;
  %When M is not square, Moore-Penrose pseudo-inverse (pinv) should be used
  Minv = inv([A; B; C; D]);
end
```

Fig. C.6 UAV motors configuration matrix.

```
function Wout = main(U, Minv)
  Wsquare = Minv * U;
  [row, col] = size(Wsquare);
  Wrpm = zeros(size(Wsquare));
  %Restriction to not cross below 0 for the speeds.
  for i = 1:row
    for j = 1:col
      if Wsquare(i, j) >= 0
        Wrps = sqrt(Wsquare(i, j));
      else
        Wrps = 0;
      end
      Wrpm(i, j) = Wrps * 60;
    end
  end
  Wout = Wrpm(:,1); %Workaround to skip the Simulink problem
end
```

Fig. C.7 Speed reference calculation block.

```
function [Va, Va_raw] = main(Wm_rpm, Wr_rpm, Wrp_rpm, Ke, Kt, Bm, Ra, Jr)
  Wm  = ((2*pi)/60)*Wm_rpm; %Convert Wm_rpm, Wr_rpm and Wrp_rpm to radians
  Wr  = ((2*pi)/60)*Wr_rpm;
  Wrp = ((2*pi)/60)*Wrp_rpm;
  k1 = 3200; %Motor control gain for simulation
  z1 = Wr - Wm; %Speed error calculation
  Va_raw = (Jr*Ra/Kt)*(Wrp + Kt*Ke*Wm/(Jr*Ra) - Bm*Wm/Jr + k1*z1);
  Va = Va_raw;
  [row col] = size(Va);
  for Index = 1:row
    if (Va(Index) > 1)
       Va(Index) = 1;
    elseif (Va < 0)
       Va(Index) = 0;
    end
  end
end
```

Fig. C.8 Motor speed control block.

```
function omega = main(W_rps, FConfig)
  calculation = 0; % Default value in case no configuration is found
  % Calculate omega taking CW motor rotations as (+) and CCW as (-)
  if (length(W_rps) == 4)
    switch FConfig
      case 0 % (0) Bouabdallah's paper
        calculation = -W_rps(1) + W_rps(2) - W_rps(3) + W_rps(4);
      case 1 % (1) Pixhawk quad X, (2) QAV 250 quad
      case 2
        calculation = -W_rps(1) - W_rps(2) + W_rps(3) + W_rps(4);
    end
  end
  omega = calculation;
end
```

Fig. C.9 Ω parameter calculation for a quad-rotor in $+$ and $\times$ configuration.

D. UAV CONTROLLER SOURCE CODE

```
// ADC to current conversion definitions
#define MULTIPLY_FACTOR   50 // ADC to current multiplication factor
#define MAX_ADC_VALUE   1023 // Maximum number the ADC of 10-bit resolution
#define MA_FACTOR       1000 // mA conversion factor
#define MA_FACTOR_MID    500 // mA conversion factor (upper)
/*************************************************************************/
/* Raw ADC conversion output to mA conversion function                   */
/*************************************************************************/
void
AdcToMACurrent (
  MOTOR_PARAMETERS MotorParameters[MAX_MOTORS],
  double           CurrentmAh[MAX_MOTORS]
  )
{
  uint8_t Index;
  double  MeanAdcValue;
  for (Index = 0; Index < MAX_MOTORS; Index++) {
    // The AdcCurrentValue contains the summ of 50 samples
    MeanAdcValue = (MotorParameters[Index].AdcCurrentValue /
                            NUM_ADC_SAMPLES);
    CurrentmAh[Index] = (MULTIPLY_FACTOR * (((MA_FACTOR *
                         MeanAdcValue) / MAX_ADC_VALUE) - MA_FACTOR_MID));
  }
}
```

Fig. D.1 ADC output to mA conversion routine.

```
// Raw counts to RPM conversion definitions
#define SECOND_IN_MICRO_SECOND_UNIT 10000000 //1 second expressed in 0.1 us #define
TIMER_PERIOD                         5 //Timer count period in 0.1 us
#define NUM_COUNTS_IN_RPM                  40 //Number of encoder markers
#define SECONDS_IN_MINUTE                  60 //Number of seconds in a minute
/*************************************************************************/
/* Raw timer counts to RPM conversion function                           */
/*************************************************************************/
void
CountsToRpm (
  MOTOR_PARAMETERS MotorParameters[MAX_MOTORS],
  double           Wm_rpm_unfiltered[MAX_MOTORS]
  )
{
  uint8_t  Index;
  uint16_t NumberOfTicks = 0;
  double   Rps;
  for (Index = 0; Index < MAX_MOTORS; Index++) {
    NumberOfTicks = (MotorParameters[Index].MotorSpeedTicks == 0) ? 65535 :
                     MotorParameters[Index].MotorSpeedTicks;
    Rps = (SECOND_IN_MICRO_SECOND_UNIT / (TIMER_PERIOD * NumberOfTicks *
           NUM_COUNTS_IN_RPM));
    Wm_rpm_unfiltered[Index] = (double)(Rps * SECONDS_IN_MINUTE);
  }
}
```

Fig. D.2 Raw timer counts to RPM speed conversion routine.

```
/*****************************************************************************/
/* Main function for performing integrals calculations                       */
/*****************************************************************************/
void
Integrate (
  double *Result, double *OldResult, double OldValue
  )
{
  // Rectangular integrator
  *Result = *OldResult + STEP_SIZE*OldValue;
  // Update OldResult to be the Result just obtained
  *OldResult = *Result;
}
```

Fig. D.3 Integrator routine code.

```
/*****************************************************************************/
/* Sign function definition                                                  */
/*****************************************************************************/
double
Sign (
  double Value
  )
{
  /* Sign function definition:
           ^____1
           |
      ---0----->
    -1____|       */
  if (Value > 0) return 1;
  else if (Value == 0) return 0;
  else return -1;
}
```

Fig. D.4 Sign function code.

```
/*****************************************************************************/
/* Function that integrates the ds0, ds1 and ds2 results of the Levant  */
/* derivative for the states array X                                         */
/*****************************************************************************/
double
IntegrateLevantDiff (
  double ds0, double ds1, double ds2, double *s0, double *s1, double *s2,
  uint8_t Index2, uint8_t ClearStatic
  )
{
         uint8_t Index;
         double  *Result[LEVANT_ORDER+1] = {s0, s1, s2};;
  static double  OldResult[LEVANT_ORDER+1] = {0};
         double  OldValue[LEVANT_ORDER+1] = {ds0, ds1, ds2};
  Result[0] = s0;
  Result[1] = s1;
  Result[2] = s2;
  // OldValue receives the derivatives of Levant to be used in the
  // integration step
  for (Index = 0; Index < LEVANT_ORDER+1; Index++) {
    // Perform integration step
    Integrate (Result[Index], &OldResult[Index][Index2], OldValue[Index]);
  }
  // Clear the static variables if indicated
  if (ClearStatic) {
    memset (&OldResult, 0, sizeof (OldResult));
  }
}
```

Fig. D.5 Levant's integrators routine code.

```
/*************************************************************************/
/* Levant's derivative for the states array X                            */
/*************************************************************************/
void
LevantDiff (
  double  X[MAX_STATES], // Input/Output
  uint8_t ClearStatic    // Input
  )
{
        uint8_t Index; double  v0, v1;
  static double  s0[MAX_STATES/2] = {0}, s1[MAX_STATES/2] = {0};
  static double  s2[MAX_STATES/2] = {0};
        double  ds0, ds1, ds2;
  for (Index = 0; Index < MAX_STATES/2; Index++) {
    // Use Index*2 in X to address 0, 2, 4, 6, 8 and 10 indexes, that
    // correspond to Roll, Pitch, Yaw, Z, X and Y, respectively
    v0  = -LAMBDA*pow (fabs (s0[Index] - X[Index*2]), POWER_2_3) *
          Sign (s0[Index] - X[Index*2]) + s1[Index];
    v1  = -LAMBDA*sqrt (fabs (s1[Index] - v0)) * Sign(s1[Index] - v0) +
          s2[Index];
    ds0 = v0;
    ds1 = v1;
    ds2 = -ALPHA*Sign (s2[Index] - v1);
    // Assign s1 to X at this point, in the next execution, the s1 is
    // be the Levant's derivative produced in the following steps
    X[Index*2 + 1] = s1[Index];
    // Update the OldValues to be ds0, ds1 and ds2 to use them in the
    // integration step
    IntegrateLevantDiffX (ds0,ds1,ds2, &s0[Index], &s1[Index], &s2[Index],
                          Index, ClearStatic);
  }
  // Clear the static variables if indicated
  if (ClearStatic) {
    memset (&s0, 0, sizeof (s0));
    memset (&s1, 0, sizeof (s1));
    memset (&s2, 0, sizeof (s2));
  }
}
```

Fig. D.6 Levant's second order derivative routine.

```
/*************************************************************************/
/* Main function for performing derivatives calculations                 */
/*************************************************************************/
void
Derivate (
  double  *Result, double  Value[MAX_DERIVATIVE_POINTS],
  uint8_t DerivativePoints
  )
{
  switch (DerivativePoints) {
    case 3: // 3 point central difference
      *Result = (1/(2*STEP_SIZE))*(Value[0] - Value[2]);
      break;
    case 5: // 5 point central difference
      *Result = (1/(10*STEP_SIZE))*(1*Value[0] + Value[1] - Value[3] -
                2*Value[4]);
      break;
    case 1:
    default: // First-difference derivative
      *Result = (1/STEP_SIZE)*(Value[0] - Value[1]);
      break;
  }
}
```

Fig. D.7 Classic derivatives calculation routine.

```
/*****************************************************************************/
/* Definition of the control gains of the system                             */
/*****************************************************************************/
void
ControlGains (double k[MAX_STATES]) {
  k[0]  = 20.53; k[1]  = 20.35; // Roll
  k[2]  = 40.92; k[3]  = 7.83;  // Pitch
  k[4]  = 15;    k[5]  = 21.5;  // Yaw
  k[6]  = 9.55;  k[7]  = 9.6;   // Z
  k[8]  = 0;     k[9]  = 0;     // X
  k[10] = 0;     k[11] = 0;     // Y
}
```

Fig. D.8 Control gains definition function.

```
/*****************************************************************************/
/* Function for calculating a 4x4 matrix determinant                         */
/*****************************************************************************/
double
DetMatrix4x4 (void) {
  // Sign rule for determinant calculation:       +-+-+-...
  //                                         Det = -+-+-+...
  //                                               +-+-+-...
  return (CalcDet4x4Element (0, 0) - CalcDet4x4Element (1, 0) +
          CalcDet4x4Element (2, 0) - CalcDet4x4Element (3, 0));
}
```

Fig. D.9 Function that calculates the determinant of a 4×4 matrix.

```
/*****************************************************************************/
/* Function for calculating a 4x4 matrix determinant element                 */
/*****************************************************************************/
double
CalcDet4x4Element (
  uint8_t Row,
  uint8_t Column
  )
{
  double Matrix3x3[3][3] = {0};
  double Element = 0;
  // Create a new matrix by skipping the row and column indicated
  FillMatrix3x3 (Row, Column, Matrix3x3);
  // Calculate the determinant of the 3x3 matrix previously generated and
  // multiply it by the M matrix element indicated
  Element = Matrix[Row][Column] * DetMatrix3x3 (Matrix3x3);
  return Element;
}
```

Fig. D.10 Routine that calculates a determinant element for the sum in the complete determinant function.

```
/*************************************************************************/
/* Function for calculating a 3x3 matrix determinant                     */
/*************************************************************************/
double
DetMatrix3x3 (
  double Matrix3x3[3][3]
  )
{
  double Determinant;

  Determinant = (Matrix3x3[0][0] * Matrix3x3[1][1] * Matrix3x3[2][2]) +
                (Matrix3x3[1][0] * Matrix3x3[2][1] * Matrix3x3[0][2]) +
                (Matrix3x3[2][0] * Matrix3x3[0][1] * Matrix3x3[1][2]) +
                (-1)*(Matrix3x3[0][2] * Matrix3x3[1][1] * Matrix3x3[2][0]) +
                (-1)*(Matrix3x3[1][2] * Matrix3x3[2][1] * Matrix3x3[0][0]) +
                (-1)*(Matrix3x3[2][2] * Matrix3x3[0][1] * Matrix3x3[1][0]);
  return Determinant;
}
```

Fig. D.11Determinant calculation function for a 3 × 3 matrix.

```
/*************************************************************************/
/* Function for calculating a the cofactor matrix of a 4x4 matrix        */
/*************************************************************************/
void
CofMatrix4x4 (double Matrix4x4[4][4]) {
  uint8_t Index1, Index2, Counter = 0;
  double  Multiplier;
  double  Matrix3x3[3][3];

  for (Index1 = 0; Index1 < 4; Index1++) {
    for (Index2 = 0; Index2 < 4; Index2++) {
      // Fill the matrix with the 3x3 matrix, skipping the row
      // and column indicated
      FillMatrix3x3 (Index1, Index2, Matrix3x3);

      // If the counter is odd, use the negative
      Multiplier = (Counter % 2) ? -1 : 1;
      //If Index1 is even, invert the previous
      if (Index1 % 2) {
        Multiplier = (Counter % 2) ? 1 : -1;
      }
      Counter++;
      // Generate the cofactor matrix
      Matrix4x4[Index1][Index2] = Multiplier * DetMatrix3x3 (Matrix3x3);
    }
  }
}
```

Fig. D.12Cofactors matrix calculation for a 4 × 4 matrix.

```
/*************************************************************************/
/* Function for calculating a the transpose matrix of a 4x4 matrix       */
/*************************************************************************/
void
TransMatrix4x4 (
  double Matrix4x4[4][4]
  )
{
  uint8_t Index1;
  uint8_t Index2;
  double  TempMatrix4x4[4][4];

  // Copy the content of the cofactor matrix into a temporary variable
  memcpy (&TempMatrix4x4, Matrix4x4, sizeof (TempMatrix4x4));

  for (Index1 = 0; Index1 < 4; Index1++) {
    for (Index2 = 0; Index2 < 4; Index2++) {
      // Invert the content to generate the transpose
      Matrix4x4[Index2][Index1] = TempMatrix4x4[Index1][Index2];
    }
  }
}
```

Fig. D.13Transpose matrix calculation Cofactors matrix calculation for a 4 × 4 matrix.

```
/*************************************************************************/
/* Function for calculating a the adjoint of a 4x4 matrix                */
/*************************************************************************/
void
CalcAdjMatrix4x4 (double Matrix4x4[4][4]) {
  memcpy (Matrix4x4, Matrix, sizeof (Matrix));
  CofMatrix4x4 (Matrix4x4);
  TransMatrix4x4 (Matrix4x4);
}
```

Fig. D.14Adjoint matrix calculation for a 4 × 4 matrix.

```
/*************************************************************************/
/* Function for calculating the inverse matrix                           */
/*************************************************************************/
void
CalcInvMatrix (double InvMatrix[MAX_MOTORS][CONTROLLER_SIGNALS]) {
  uint8_t Index1, Index2; double Det; double AdjMatrix4x4[4][4];
  if (MAX_MOTORS == 4) {
    Det = DetMatrix4x4 ();
    CalcAdjMatrix4x4 (AdjMatrix4x4);
    for (Index1 = 0; Index1 < MAX_MOTORS; Index1++) {
      for (Index2 = 0; Index2 < CONTROLLER_SIGNALS; Index2++) {
        InvMatrix[Index1][Index2] = AdjMatrix4x4[Index1][Index2]/Det;
      }
    }
  }
}
```

Fig. D.15Inverse matrix calculation algorithm for a 4 × 4 matrix.

```
/*********************************************************************/
/* Function that generates the reference signals for trajectory tracking*/
/*********************************************************************/
void
ReferenceSignalsTrajectory (
  double References[CONTROLLER_SIGNALS], double *Counter
  )
{
  double  Yaw = 0, Zr = 3, Xr = 1, Yr = 2; uint8_t Time[4] = {4, 8, 12, 16};
  References[0] = Yaw;
  References[1] = Zr/pow(1+exp(-SLOPE*(*Counter -(5/SLOPE))), 2);
  References[2] = Xr/pow(1+exp(-SLOPE*(*Counter -Time[0] - (5/SLOPE))), 2) -
                  Xr/pow(1+exp(-SLOPE*(*Counter -Time[2] - (5/SLOPE))), 2);
  References[3] = Yr/pow(1+exp(-SLOPE*(*Counter -Time[1] - (5/SLOPE))), 2) -
                  Yr/pow(1+exp(-SLOPE*(*Counter -Time[3] - (5/SLOPE))), 2);
  *Counter += STEP_SIZE;
}
```

Fig. D.16Trajectory tracking reference generation for controllable states.

```
/*********************************************************************/
/* Function that generates all the reference states array for the       */
/* trajectory flight mode                                               */
/*********************************************************************/
void
GenerateXrTrajectory (
  double X1r, double X3r, double References[CONTROLLER_SIGNALS],
  double Xr[MAX_STATES], double Xrp[MAX_STATES],
  uint8_t UseLevantDerivatives, uint8_t DerivativePoints,uint8_t ClearStatic
  )
{
  // Fill the Xr array with the actual roll, pitch, yaw, Z, X and Y values
  Xr[0] = X1r; Xr[2] = X3r; Xr[4] = References[0]; Xr[6] = References[1];
  Xr[8] = References[2]; Xr[10] = References[3];
  if (UseLevantDerivatives) {
    // Derivate the reference values, then derivate Xr to obtain Xrp
    LevantDiffReference (Xr, 0); LevantDiffXr (Xr, Xrp, 0);
  } else {
    // Derivate the reference values, then derivate Xr to obtain Xrp
    DerivateReference (Xr, DerivativePoints, ClearStatic);
    DerivateXr (Xr, Xrp, DerivativePoints, ClearStatic);
  }
}
```

Fig. D.17Function that generates the 6 temporary variables references array ($\phi, \theta, \psi, z, x,$ and y) and obtains the first and second derivatives for X_r and $\dot{X}_r$.

```
/*************************************************************************/
/* Error calculation of the states in the system                         */
/*************************************************************************/
void
UAVErrorCalculation (
  double  X[MAX_STATES],   // Input
  double  Xr[MAX_STATES],  // Input
  double  XrV[MAX_STATES], // Output
  double  Xrp[MAX_STATES], // Input
  double  k[MAX_STATES],   // Input
  double  Z[MAX_STATES],   // Output
  double  Zp[MAX_STATES],  // Output
  uint8_t DirectControl    // Input
  )
{
  /* Roll angle error */
  Z[0]    = Xr[0] - X[0];         // Z1    = X1r - X1
  Zp[0]   = Xr[1] - X[1];         // Z1p   = X2r - X2
  XrV[1]  = Xr[1] + k[0]*Z[0];    // X2r*  = X2r + k1*Z1
  Z[1]    = XrV[1] - X[1];        // Z2    = X2r* - X2

  /* Pitch angle error */
  Z[2]    = Xr[2] - X[2];         // Z3    = X3r - X3
  Zp[2]   = Xr[3] - X[3];         // Z3p   = X4r - X4
  XrV[3]  = Xr[3] + k[2]*Z[2];    // X4r*  = X4r + k3*Z3
  Z[3]    = XrV[3] - X[3];        // Z4    = X4r* - X4

  /* Yaw angle error */
  Z[4]    = Xr[4] - X[4];         // Z5    = X5r - X5
  Zp[4]   = Xr[5] - X[5];         // Z5p   = X6r - X6
  XrV[5]  = Xr[5] + k[4]*Z[4];    // X6r*  = X6r + k5*Z5
  Z[5]    = XrV[5] - X[5];        // Z6    = X6r* - X6

  /* Z position error */
  Z[6]    = Xr[6] - X[6];         // Z7    = X7r - X7
  Zp[6]   = Xr[7] - X[7];         // Z7p   = X8r - X8
  XrV[7]  = Xr[7] + k[6]*Z[6];    // X8r*  = X8r + k7*Z7
  Z[7]    = XrV[7] - X[7];        // Z8    = X8r* - X8

  /* X position error */
  Z[8]    = Xr[8] - X[8];         // Z9    = X9r - X9
  Zp[8]   = Xr[9] - X[9];         // Z9p   = X10r - X10
  XrV[9]  = Xr[9] + k[8]*Z[8];    // X10r* = X10r + k9*Z9
  Z[9]    = XrV[9] - X[9];        // Z10   = X10r* - X10

  /* Y position error */
  Z[10]   = Xr[10] - X[10];       // Z11   = X11r - X11
  Zp[10]  = Xr[11] - X[11];       // Z11p  = X12r - X12
  XrV[11] = Xr[11] + k[10]*Z[10]; // X12r* = X12r + k11*Z11
  Z[11]   = XrV[11] - X[11];      // Z12   = X12r* - X12
}
```

Fig. D.18Error calculation function, that also generates the virtual error.

```
/*****************************************************************************/
/* Function that calculates Omega parameter based on the motors RPS and */
/* frame configuration                                                       */
/*****************************************************************************/
void
OmegaCalculation (
  double W_rps[MAX_MOTORS],
  double *Omega
  )
{
  // Default value in case no configuration is found
  *Omega = 0;
  switch (FRAME_CONFIGURATION) {
    // Bouabdallah's paper
    case '+':
      *Omega = -W_rps[0] + W_rps[1] - W_rps[2] + W_rps[3];
      break;
    // Pixhawk quad X and QAV 250 configuration
    case 'X':
    case 'Q':
      *Omega = -W_rps[0] - W_rps[1] + W_rps[2] + W_rps[3];
      break;
  }
}
```

Fig. D.19Block that calculates the angular speed difference Ω from Table 1.1.

```
/*****************************************************************************/
/* Function for calculating the reference speed                              */
/*****************************************************************************/
void
CalcWr (
  double U[CONTROLLER_SIGNALS],
  double InvMatrix[MAX_MOTORS][CONTROLLER_SIGNALS],
  double Wr_rps[MAX_MOTORS],
  double Wr_rpm[MAX_MOTORS]
  )
{
  uint8_t Index1;
  uint8_t Index2;
  double  Speed;

  for (Index1 = 0; Index1 < MAX_MOTORS; Index1++) {
    // Calculate the multiplication with U
    Speed = 0;
    for (Index2 = 0; Index2 < CONTROLLER_SIGNALS; Index2++) {
      Speed += InvMatrix[Index1][Index2] * U[Index2];
    }

    // Calculate the square root of the speed
    if (Speed >= 0) {
      // U = sum(F) = M^-1*W where W is in RPS units
      Wr_rps[Index1] = sqrt (Speed);
      // Multiply Wr_rps by 60 to convert RPS to RPM, which is what
      // F = Ct*Rho*W_rps^2*PropDiameter^4 uses
      Wr_rpm[Index1] = Wr_rps[Index1]*60;
    } else {
      Wr_rps[Index1] = 0;
      Wr_rpm[Index1] = 0;
    }
  }
}
```

Fig. D.20Angular speed reference processing function.

```
/*************************************************************************/
/* UAV Backstepping linear controller code for trajectory flight mode   */
/*************************************************************************/
void
BacksteppingLinearControlTrajectory (
  double Mass,
  double k[MAX_STATES],
  double Omega,
  double X[MAX_STATES],
  double Xrp[MAX_STATES],
  double Z[MAX_STATES],
  double Zp[MAX_STATES],
  double U[CONTROLLER_SIGNALS], // Output
  double *X1r,                  // Output
  double *X3r                   // Output
  ) {
  double Ux, Uy, Temp;
  // DERIVATIVES
  // Xrp[1]  = Xrpp[0]  - Roll | Xrp[3]  = Xrpp[2]  - Pitch
  // Xrp[5]  = Xrpp[4]  - Yaw  | Xrp[7]  = Xrpp[6]  - Z
  // Xrp[9]  = Xrpp[8]  - X    | Xrp[11] = Xrpp[10] - Y
  /* Yaw angle control (U4 calculation) */
  U[3] = (1 / b_var2)*(Xrp[5] + k[4]*Zp[4] - X[1]*X[3]*a_var4 + Z[4] +
                       k[5]*Z[5]);
  /* Z position control (U1 calculation) */
  U[0] = (Mass / (cos (X[0])*cos (X[2])))*(Xrp[7] + k[6]*Zp[6] + Gravity +
                                           Z[6] + k[7]*Z[7]);
  /* X position control */
  Ux = (Mass / U[0])*(Xrp[9] + k[8]*Zp[8] + Z[8] + k[9]*Z[9]);
  /* Y position control */
  Uy = (Mass / U[0])*(Xrp[11] + k[10]*Zp[10] + Z[10] + k[11]*Z[11]);
  /* Roll angle control (U2 calculation) */
  U[1] = (1 / b_var0)*(Xrp[1] + k[0]*Zp[0] - X[3]*X[5]*a_var0 -
                       X[3]*a_var1*Omega + Z[0] + k[1]*Z[1]);
  Temp = (Ux*tan (X[4]) - Uy) / (sin (X[4])*tan (X[4]) + cos (X[4]));
  if (Temp > 1) { // Sanity check for arcsin function definition boundaries
    Temp = 1;
  } else if (Temp < -1) {
    Temp = -1;
  }
  *X1r = asin (Temp);
  /* Pitch angle control (U3 calculation) */
  U[2] = (1 / b_var1)*(Xrp[3] + k[2]*Zp[2] - X[1]*X[5]*a_var2 -
                       X[1]*a_var3*Omega + Z[2] + k[3]*Z[3]);
  Temp = (Ux - sin (X[0])*sin (X[4])) / (cos (X[0])*cos (X[4]));
  if (Temp > 1) { // Sanity check for arcsin function definition boundaries
    Temp = 1;
  } else if (Temp < -1) {
    Temp = -1;
  }
  *X3r = asin (Temp);
}
```

Fig. D.21Backstepping with linear control code for trajectory tracking flight mode.

```
/*************************************************************************/
/* UAV backstepping motor controller code                                */
/*************************************************************************/
void
MotorControlBackstepping (
  double Z_radps[MAX_MOTORS],   // Input
  double Wr_radps[MAX_MOTORS],  // Input
  double Wrp_radps[MAX_MOTORS], // Input
  double Wm_radps[MAX_MOTORS],  // Input
  double ControlGain,           // Input
  double Va[MAX_MOTORS],        // Output
  double Va_raw[MAX_MOTORS]     // Output
  )
{
  uint8_t Index;
  double  k1 = ControlGain;

  for (Index = 0; Index < MAX_MOTORS; Index++) {
    // Calculation of the voltage applied in the motor
    Va_raw[Index] = (Jr*Ra/Kt)*(Wrp_radps[Index] + Kt*Ke*Wm_radps[Index]
                    /(Jr*Ra) - Bm*Wm_radps[Index]/Jr + k1*Z_radps[Index]);
    Va[Index] = Va_raw[Index];

    // Sanity check for the voltage (PWM duty cycle)
    if (Va[Index] > 1) {
      Va[Index] = 1;
    } else if (Va[Index] < 0) {
      Va[Index] = 0;
    }
  }
}
```

Fig. D.22Motor control algorithm.

```
/*************************************************************************/
/* EWMA motor speed filter for the noise from the speed sensors          */
/*************************************************************************/
void
SpeedFilterEWMA (
  double W_rpm_in[MAX_MOTORS], // Input
  double W_rpm_out[MAX_MOTORS] // Output
  )
{
  uint8_t Index;
  for (Index = 0; Index < MAX_MOTORS; Index++) {
    W_rpm_out[Index] = W_rpm_out[Index] + LPHA_SPEED_FILTER*(W_rpm_in[Index]
                       - W_rpm_out[Index]);
  }
}
```

Fig. D.23EWMA speed filter routine.

```
/*************************************************************************/
/* Functions that executes all the controller code involved              */
/*************************************************************************/
void
ControllerExecution (
  double *Timer,
  double Mass,
  double InvMatrix[MAX_MOTORS][CONTROLLER_SIGNALS],
  double k[MAX_STATES],        //Input
  double kDv[MAX_STATES/2],    //Input
  double X[MAX_STATES],        //Input
  double Xr[MAX_STATES],       //Output
  double XrV[MAX_STATES],      //Output
  double Xrp[MAX_STATES],      //Output
  double Z[MAX_STATES],        //Output
  double Zp[MAX_STATES],       //Output
  double Wr_rps[MAX_MOTORS],   //Output
  double Wr_rpm[MAX_MOTORS],   //Output
  double Wrp_rpm[MAX_MOTORS],  //Output
  double Wm_rps[MAX_MOTORS],   //Input
  double Wm_rpm[MAX_MOTORS],   //Input
  double *Omega,               //Output
  double MotorControlGain,     //Input
  double MotorControlDGain,    //Input
  double Va[MAX_MOTORS],       //Output
  double U[CONTROLLER_SIGNALS] //Output
  )
{
  double References[CONTROLLER_SIGNALS] = {0};
  // Motor control variables
  double Wr_radps[MAX_MOTORS] = {0}, Wrp_radps[MAX_MOTORS] = {0};
  double Wm_radps[MAX_MOTORS] = {0}, Z_motor_radps[MAX_MOTORS] = {0};
  double Z_motor_rpm[MAX_MOTORS] = {0}, Va_raw[MAX_MOTORS] = {0};
  // Control variables
#if (FLIGHT_MODE == FLIGHT_MODE_TRAJECTORY)
  static double X1r = 0;
  static double X3r = 0;
#endif // FLIGHT_MODE_TRAJECTORY

  // Generate the reference signals for testing
#if (FLIGHT_MODE == FLIGHT_MODE_STABILIZE)
  ReferenceSignalsStabilize (References, Timer);
#elif (FLIGHT_MODE == FLIGHT_MODE_TRAJECTORY)
  ReferenceSignalsTrajectory (References, Timer);
#endif // FLIGHT_MODE
  // Generate the complete array of states
  // Use X[9] and X[11] since this controller does not generate X and Y
#if (FLIGHT_MODE == FLIGHT_MODE_STABILIZE)
  // X[8] and X[10] provide the references for X and Y
  GenerateXrStabilize (References, X[8], X[10], Xr, Xrp, 0, 1, 0);
#elif (FLIGHT_MODE == FLIGHT_MODE_TRAJECTORY)
  // X1r and X3r provide the references for Roll and Pitch
  GenerateXrTrajectory (X1r, X3r, References, Xr, Xrp, 0, 1, 0);
#endif // FLIGHT_MODE
  // With Xr and X defined, calculate the error between them
  UAVErrorCalculation (X, Xr, XrV, Xrp, k, Z, Zp, 0);

  // Calculate the total speed combination
  OmegaCalculation (Wm_rps, Omega);

  // Call the controller block for obtaining the control vector U
#if (FLIGHT_MODE == FLIGHT_MODE_STABILIZE)

#if (UAV_CONTROLLER_TYPE == BACKSTEPPING_CONTROLLER)
  BackSteppingLinearControlStabilize (Mass, k, *Omega, X, Xrp, Z, Zp, U);
#elif (UAV_CONTROLLER_TYPE == SLIDING_MODE_CONTROLLER)
  SlidingModeControlStabilize (Mass, k, *Omega, X, Xrp, Z, Zp, U);
#elif (UAV_CONTROLLER_TYPE == SUPER_SWISTING_CONTROLLER)
  SuperTwistingControlStabilize (Mass, k, kDv, *Omega, X, Xrp, Z, Zp, U);
#endif // UAV_CONTROLLER_TYPE
```

```
#elif (FLIGHT_MODE == FLIGHT_MODE_TRAJECTORY)

#if (UAV_CONTROLLER_TYPE == BACKSTEPPING_CONTROLLER)
  BacksteppingLinearControlTrajectory (Mass, k, *Omega, X, Xrp, Z, Zp, U, &X1r, &X3r);
#elif (UAV_CONTROLLER_TYPE == SLIDING_MODE_CONTROLLER)
  SlidingModeControlTrajectory (Mass, k, *Omega, X, Xrp, Z, Zp, U, &X1r, &X3r);
#elif (UAV_CONTROLLER_TYPE == SUPER_SWISTING_CONTROLLER)
  SuperTwistingControlTrajectory (Mass, k, kDv, *Omega, X, Xrp, Z, Zp, U, &X1r, &X3r);
#endif // UAV_CONTROLLER_TYPE

#endif // FLIGHT_MODE

  // Calculate the speed reference with the control vector U and the inverse
  // matrix InvM
  CalcWr (U, InvMatrix, Wr_rps, Wr_rpm);

  // Calculate the derivative of Wr_rpm to be used in the motor control
  // algorithm
  LevantDiffWr (Wr_rpm, Wrp_rpm, 0);

  // Convert RPMs to radians per second
  MotorRpmToRadps (Wr_rpm, Wrp_rpm, Wm_rpm, Wr_radps, Wrp_radps, Wm_radps);

  // Motor speeds error calculation
  MotorSpeedError (Wr_radps, Wm_radps, Z_motor_radps, Z_motor_rpm);

  // With the angular speed reference, the next step is to calculate the
  // motor control for obtaining the voltage (PWM duty cycle) to be applied
  // in the voltage-current and voltage-speed transfer functions
  MotorControlBackstepping (Z_motor_radps, Wr_radps, Wrp_radps, Wm_radps,
                            MotorControlGain, Va, Va_raw);
}
```

Fig. D.24UAV controller flow.

```
/*********************************************************************/
/* Voltage-speed transfer function in state-space shape              */
/*********************************************************************/
void
SpeedTranferFunction (
  double  U[MAX_MOTORS],  //Input
  double  X1[MAX_MOTORS], //Input/Output
  double  X2[MAX_MOTORS], //Input/Output
  double  Y[MAX_MOTORS],  //Output
  uint8_t ClearStatic     //Input
  )
{
  uint8_t Index;
  double  X1p;
  double  X2p;

  for (Index = 0; Index < MAX_MOTORS; Index++) {
    // Speed derivative
    X1p = -(double)11.16*X1[Index] - (double)0.4724*X2[Index] + U[Index];
    // Current derivative
    X2p = 1*X1[Index];

    Y[Index] = (double)175098.7044*X1[Index] + (double)9082.095516*X2[Index] +
(double)967.41*U[Index];

    IntegrateSpeedX1p (X1p, &X1[Index], Index, ClearStatic);
    IntegrateSpeedX2p (X2p, &X2[Index], Index, ClearStatic);
  }
}
```

Fig. D.25Voltage to speed transfer function routine.

```
/*************************************************************************/
/* Function that calculates the states of the UAV system with its        */
/* mathematical model                                                    */
/*************************************************************************/
void
MathematicalModelUAV (
  double Mass, double X[MAX_STATES], double Xr[MAX_STATES],
  double W_rps[MAX_MOTORS], double Omega
  )
{
        uint8_t Index;
        double  Xp[MAX_STATES] = {0};
        double  Matrix[CONTROLLER_SIGNALS][MAX_MOTORS];
        double  U[CONTROLLER_SIGNALS]={0},W_rps_square[MAX_MOTORS], Ux, Uy;
  static double  XTemp[MAX_STATES] = {0};
  GetMatrix (Matrix);
  // Obtain the square of the speed using RPS
  for (Index = 0; Index < MAX_MOTORS; Index++) {
    W_rps_square[Index] = pow (W_rps[Index], 2);
  }
  // Assign the integrated values of the states from the previous
  // execution to X, to be used in the calculations of this run
  for (Index = 0; Index < MAX_STATES; Index++) {
    X[Index] = XTemp[Index];
  }
  // Obtain the controller signals
  CalcU (Matrix, W_rps_square, U);
  Ux = (cos (X[0])*sin (X[2])*cos (X[4]) + sin (X[0])*sin (X[4]));
  Uy = (cos (X[0])*sin (X[2])*sin (X[4]) - sin (X[0])*cos (X[4]));
  //Roll
  Xp[0]  = X[1];
  Xp[1]  = X[3]*X[5]*a_var0 + X[3]*a_var1*Omega + b_var0*U[1];
  //Pitch
  Xp[2]  = X[3];
  Xp[3]  = X[1]*X[5]*a_var2 + X[1]*a_var3*Omega + b_var1*U[2];
  //Yaw
  Xp[4]  = X[5];
  Xp[5]  = X[3]*X[1]*a_var4 + b_var2*U[3];
  //Z
  Xp[6]  = X[7];
  Xp[7]  = -Gravity + (cos (X[0])*cos (X[2]))*(1/Mass)*U[0];
  //X
  Xp[8]  = X[9];
  Xp[9]  = Ux*(1/Mass)*U[0];
  //Y
  Xp[10] = X[11];
  Xp[11] = Uy*(1/Mass)*U[0];
  // Integrate Xp
  for (Index = 0; Index < MAX_STATES; Index++) {
    IntegrateXp (Xp[Index], &XTemp[Index], Index, 0);
  }
}
```

Fig. D.26Multi-rotor mathematical model code.

E. SOURCE CODE OF THE GUI FOR SYSTEM MONITORING

```
// Populate the serial COMs drop-down box
SerialPort[] portNames = SerialPort.getCommPorts();
for (Index = 0; Index < portNames.length; Index++) {
  PortList.addItem(portNames[Index].getSystemPortName());
}
// Configure the connect button and use another thread to listen for data
ConnectButton.addActionListener(new ActionListener() {
  @Override public void actionPerformed(ActionEvent arg0) {
    if (ConnectButton.getText().equals("Connect")) {
      // Attempt to connect to the serial port
      ChosenPort = SerialPort.getCommPort(
                    PortList.getSelectedItem().toString());
      ChosenPort.setComPortTimeouts(SerialPort.TIMEOUT_SCANNER, 0, 0);
      // BAUD_RATE, 8 bits to transmit, 1 stop bit, no parity bits
      ChosenPort.setComPortParameters(BAUD_RATE, 8, 1, 0);
      if (ChosenPort.openPort()) {
        ConnectButton.setText("Disconnect");
        PortList.setEnabled(false);
      }
    } else {
      // Disconnect from the serial port
      ChosenPort.closePort();
      PortList.setEnabled(true);
      // Restore the button string to Run if pressed "Stop"
      ConnectButton.setText("Connect");
    }
  }
});
```

Fig. E.1 Serial communication initialization in the GUI.

```
// Create a new thread that listens for incoming text and graphs the data
Thread ThreadRx = new Thread() {
  @Override public void run() {
    PrintGraph (
      SpeedsDataset1, SpeedsDataset2, SpeedsDataset3, SpeedsDataset4,
      CurrentsDataset,
      AnglesDataset1, AnglesDataset2, AnglesDataset3,
      PositionsDataset1, PositionsDataset2, PositionsDataset3,
      ControlOutputDataset,
      MotorControlOutputDataset,
      ExtraData,
      SpeedMonitoringChart1, SpeedMonitoringChart2,
      SpeedMonitoringChart3, SpeedMonitoringChart4,
      CurrentMonitoringChart,
      AnglesMonitoringChart1, AnglesMonitoringChart2,AnglesMonitoringChart3,
      PositionsMonitoringChart1, PositionsMonitoringChart2,
      PositionsMonitoringChart3, Window);
  }
};
ThreadRx.start();// Start the previously created thread
```

Fig. E.2 Function definition of the serial communication thread for receiving data and plot it, created within the UART button connection listener.

```
/*****************************************************************/
/* Routine to display the graph with the reference and scanned data     */
/*****************************************************************/
public static void
PrintGraph (
  XYSeriesCollection SpeedsDataset1, XYSeriesCollection SpeedsDataset2,
  XYSeriesCollection SpeedsDataset3, XYSeriesCollection SpeedsDataset4,
  XYSeriesCollection CurrentsDataset,
  XYSeriesCollection AnglesDataset1, XYSeriesCollection AnglesDataset2,
```

```
  XYSeriesCollection AnglesDataset3,
  XYSeriesCollection PositionsDataset1, XYSeriesCollection PositionsDataset2,
  XYSeriesCollection PositionsDataset3,
  XYSeriesCollection ControlOutputData,
  XYSeriesCollection MotorControlOutputData,
  XYSeriesCollection ExtraData,
  JFreeChart         SpeedMonitoringChart1, JFreeChart SpeedMonitoringChart2,
  JFreeChart         SpeedMonitoringChart3, JFreeChart SpeedMonitoringChart4,
  JFreeChart         CurrentMonitoringChart,
  JFreeChart         AnglesMonitoringChart1, JFreeChart AnglesMonitoringChart2,
  JFreeChart         AnglesMonitoringChart3,
  JFreeChart         PositionsMonitoringChart1, JFreeChart PositionsMonitoringChart2,
  JFreeChart         PositionsMonitoringChart3,
  JFrame             Window
  )
{
  Scanner  SerialData = new Scanner(ChosenPort.getInputStream());
  String[] Line;
  double[] Numbers;
  int      Index, Length, DataCounter, StartXAxisValue, EndXAxisValue;

  // The following line scans for upcoming data starting with "\n"
  while (SerialData.hasNext()) {
    try {
      // Start the data counter index to 0
      DataCounter = 0;
      // Separate the whole string into sub-strings that are divided by "," character
      Line = SerialData.nextLine().split(",");
      Length = Line.length;
      //Numbers = new int[Length];
      Numbers = new double[Length];
      // In case of receiving a data package that is different in size compared
      // to what is expected, fill the array of Numbers with zeros and do nothing
      if (Length != DATA_TO_READ) {
        //Fill with zeros
        for (Index = 0; Index < Length; Index++) {
          Numbers[Index] = 0;
        }
      } else {
        // Convert serial data into integers for plotting
        for (Index = 0; Index < Length; Index++) {
          // Number in RPM
          Numbers[Index] = Double.parseDouble(Line[Index]);
        }
        // Start the data counter index to 0
        DataCounter = 0;
        // SPEEDS DATASETS
        // Add the numbers to the speed reference dataset
        for (Index = 0; Index < SpeedsDataset1.getSeriesCount(); Index++) {
          SpeedsDataset1.getSeries(Index).add(XActual, Numbers[DataCounter]);
          DataCounter++;
        }
        // Add the numbers to the speed reference dataset
        for (Index = 0; Index < SpeedsDataset2.getSeriesCount(); Index++) {
          SpeedsDataset2.getSeries(Index).add(XActual, Numbers[DataCounter]);
          DataCounter++;
        }
        // Add the numbers to the speed reference dataset
        for (Index = 0; Index < SpeedsDataset3.getSeriesCount(); Index++) {
          SpeedsDataset3.getSeries(Index).add(XActual, Numbers[DataCounter]);
          DataCounter++;
        }
        // Add the numbers to the speed reference dataset
        for (Index = 0; Index < SpeedsDataset4.getSeriesCount(); Index++) {
          SpeedsDataset4.getSeries(Index).add(XActual, Numbers[DataCounter]);
          DataCounter++;
        }
        // Add the motor currents to the datasets
        for (Index = 0; Index < CurrentsDataset.getSeriesCount(); Index++) {
          CurrentsDataset.getSeries(Index).add(XActual, Numbers[DataCounter]);
          DataCounter++;
```

```
        }
        // ANGLES DATASETS
        // Add the Roll angle to the datasets
        for (Index = 0; Index < AnglesDataset1.getSeriesCount(); Index++) {
          AnglesDataset1.getSeries(Index).add(XActual, Numbers[DataCounter]);
          DataCounter++;
        }
        // Add the Pitch angle to the datasets
        for (Index = 0; Index < AnglesDataset2.getSeriesCount(); Index++) {
          AnglesDataset2.getSeries(Index).add(XActual, Numbers[DataCounter]);
          DataCounter++;
        }
        // Add the Yaw angle to the datasets
        for (Index = 0; Index < AnglesDataset3.getSeriesCount(); Index++) {
          AnglesDataset3.getSeries(Index).add(XActual, Numbers[DataCounter]);
          DataCounter++;
        }
        // POSITIONS DATASETS
        // Add the Z position to the datasets
        for (Index = 0; Index < PositionsDataset1.getSeriesCount(); Index++) {
          PositionsDataset1.getSeries(Index).add(XActual, Numbers[DataCounter]);
          DataCounter++;
        }
        // Add the X position to the datasets
        for (Index = 0; Index < PositionsDataset2.getSeriesCount(); Index++) {
          PositionsDataset2.getSeries(Index).add(XActual, Numbers[DataCounter]);
          DataCounter++;
        }
        // Add the Y position to the datasets
        for (Index = 0; Index < PositionsDataset3.getSeriesCount(); Index++) {
          PositionsDataset3.getSeries(Index).add(XActual, Numbers[DataCounter]);
          DataCounter++;
        }
        // CONTROLLERS DATASETS
        // Add the controllers output numbers to the datasets
        for (Index = 0; Index < ControlOutputData.getSeriesCount(); Index++) {
          ControlOutputData.getSeries(Index).add(XActual, Numbers[DataCounter]);
          DataCounter++;
        }
        // Add the motor controllers output numbers to the datasets
        for (Index = 0; Index < MotorControlOutputData.getSeriesCount(); Index++) {
          MotorControlOutputData.getSeries(Index).add(XActual, Numbers[DataCounter]);
          DataCounter++;
        }
        // Add the extra data to the dataset
        for (Index = 0; Index < ExtraData.getSeriesCount(); Index++) {
          ExtraData.getSeries(Index).add(XActual, Numbers[DataCounter]);
          DataCounter++;
        }
        XActual++; // Increase the index counter (sample number)
        // If the number of samples is higher than the specified, start setting new ranges
        // (this is for displaying the graph as cached)
        if (XActual > 1000) {
          // Get the new X axis labels for cached printing
          EndXAxis = SpeedsDataset1.getSeries(0).getX(SpeedsDataset1.getItemCount(0)-
                     1).intValue();
          StartXAxis = EndXAxisValue-1000;
          // Set the new X axis range
          SpeedMonitoringChart1.getXYPlot().getDomainAxis().setRange(StartXAxis, EndXAxis);
          SpeedMonitoringChart2.getXYPlot().getDomainAxis().setRange(StartXAxis, EndXAxis);
          SpeedMonitoringChart3.getXYPlot().getDomainAxis().setRange(StartXAxis, EndXAxis);
          SpeedMonitoringChart4.getXYPlot().getDomainAxis().setRange(StartXAxis, EndXAxis);
          CurrentMonitoringChart.getXYPlot().getDomainAxis().setRange(StartXAxis, EndXAxis);
          AnglesMonitoringChart1.getXYPlot().getDomainAxis().setRange(StartXAxis, EndXAxis);
          AnglesMonitoringChart2.getXYPlot().getDomainAxis().setRange(StartXAxis, EndXAxis);
          AnglesMonitoringChart3.getXYPlot().getDomainAxis().setRange(StartXAxis, EndXAxis);
          PositionsMonitoringChart1.getXYPlot().getDomainAxis().setRange(StartXAxis, EndXAxis);
          PositionsMonitoringChart2.getXYPlot().getDomainAxis().setRange(StartXAxis, EndXAxis);
          PositionsMonitoringChart3.getXYPlot().getDomainAxis().setRange(StartXAxis, EndXAxis);
        }
        // Write the CSV file with the data sets
```

```
        WriteToCSV (SpeedsDataset1, SpeedsDataset2, SpeedsDataset3, SpeedsDataset4,
                    CurrentsDataset, AnglesDataset1, AnglesDataset2, AnglesDataset3,
                    PositionsDataset1, PositionsDataset2, PositionsDataset3,
                    ControlOutputData, MotorControlOutputData, ExtraData);
      }
      // If the index of the datasets is greater than 1000, and the modulus 1000 is 0 (2000,
      // 3000, etc.), then clear the first 1000 elements in the datasets
      if ((XActual > 1000) && ((XActual % 1000) == 0)) {
        // Remove the first 1000 elements in the datasets
        for (Index = 0; Index < SpeedsDataset1.getSeriesCount(); Index++) {
          SpeedsDataset1.getSeries(Index).delete(0, 999);
          SpeedsDataset2.getSeries(Index).delete(0, 999);
          SpeedsDataset3.getSeries(Index).delete(0, 999);
          SpeedsDataset4.getSeries(Index).delete(0, 999);
        }
        // Remove the first 1000 elements in the dataset
        for (Index = 0; Index < CurrentsDataset.getSeriesCount(); Index++) {
          CurrentsDataset.getSeries(Index).delete(0, 999);
        }
        // Remove the first 1000 elements in the datasets
        for (Index = 0; Index < AnglesDataset1.getSeriesCount(); Index++) {
          AnglesDataset1.getSeries(Index).delete(0, 999);
          AnglesDataset2.getSeries(Index).delete(0, 999);
          AnglesDataset3.getSeries(Index).delete(0, 999);
        }
        // Remove the first 1000 elements in the datasets
        for (Index = 0; Index < PositionsDataset1.getSeriesCount(); Index++) {
          PositionsDataset1.getSeries(Index).delete(0, 999);
          PositionsDataset2.getSeries(Index).delete(0, 999);
          PositionsDataset3.getSeries(Index).delete(0, 999);
        }
        // Remove the first 1000 elements in the dataset
        for (Index = 0; Index < ControlOutputData.getSeriesCount(); Index++) {
          ControlOutputData.getSeries(Index).delete(0, 999);
        }
        // Remove the first 1000 elements in the dataset
        for (Index = 0; Index < MotorControlOutputData.getSeriesCount(); Index++) {
          MotorControlOutputData.getSeries(Index).delete(0, 999);
        }
        // Remove the first 1000 elements in the dataset
        for (Index = 0; Index < ExtraData.getSeriesCount(); Index++) {
          ExtraData.getSeries(Index).delete(0, 999);
        }
      }
    } catch (Exception ex) {
      System.out.println(ex);
      System.out.println("Error when printing graph");
    }
  }
  SerialData.close();
}
```

Fig. E.3 Serial communication and data logging thread function.

```
/************************************************************************/
/* Routine to write the data generated into a CSV file                  */
/************************************************************************/
public static void
WriteToCSV (
  XYSeriesCollection SpeedsDataset1, XYSeriesCollection SpeedsDataset2,
  XYSeriesCollection SpeedsDataset3, XYSeriesCollection SpeedsDataset4,
  XYSeriesCollection CurrentsDataset, XYSeriesCollection AnglesDataset1,
  XYSeriesCollection AnglesDataset2, XYSeriesCollection AnglesDataset3,
  XYSeriesCollection PositionsDataset1,XYSeriesCollection PositionsDataset2,
  XYSeriesCollection PositionsDataset3,XYSeriesCollection ControlOutputData,
  XYSeriesCollection MotorControlOutputData, XYSeriesCollection ExtraData
  ) {
  int Index, DataCounter, LastTimer, LastIndex;
  if (LogFile != null) {
    // Use StringBuilder instead of StringBuffer for performance and sync
```

```
    StringBuilder CsvData = new StringBuilder("");
    DataCounter = 0; // Reset the data counter index to 0
    LastTimer = SpeedsDataset1.getEndX(// Get the last index in the dataset
                0, SpeedsDataset1.getItemCount(0)-1).intValue();
    LastIndex = SpeedsDataset1.getSeries(0).getItemCount()-1;
    CsvData.append(LastTimer*STEP_SIZE); // Add the time column
    CsvData.append(",");
    // Add the motor references and speeds data
    for (Index = 0; Index < SpeedsDataset1.getSeriesCount(); Index++) {
      // Use the LastIndex to get the last speed value in the dataset
      CsvData.append(
        SpeedsDataset1.getSeries(Index).getY(LastIndex).doubleValue());
      CsvData.append(",");
    }
    for (Index = 0; Index < SpeedsDataset2.getSeriesCount(); Index++) {
      CsvData.append(
        SpeedsDataset2.getSeries(Index).getY(LastIndex).doubleValue());
      CsvData.append(",");
    }
    for (Index = 0; Index < SpeedsDataset3.getSeriesCount(); Index++) {
      CsvData.append(
        SpeedsDataset3.getSeries(Index).getY(LastIndex).doubleValue());
      CsvData.append(",");
    }
    for (Index = 0; Index < SpeedsDataset4.getSeriesCount(); Index++) {
      CsvData.append(
        SpeedsDataset4.getSeries(Index).getY(LastIndex).doubleValue());
      CsvData.append(",");
    }
    // Add the motor currents data
    for (Index = 0; Index < CurrentsDataset.getSeriesCount(); Index++) {
      CsvData.append(
        CurrentsDataset.getSeries(Index).getY(LastIndex).doubleValue());
      CsvData.append(",");
    }
    // Add the Euler angles data
    for (Index = 0; Index < AnglesDataset1.getSeriesCount(); Index++) {
      CsvData.append(
        AnglesDataset1.getSeries(Index).getY(LastIndex).doubleValue());
      CsvData.append(",");
    }
    for (Index = 0; Index < AnglesDataset2.getSeriesCount(); Index++) {
      CsvData.append(
        AnglesDataset2.getSeries(Index).getY(LastIndex).doubleValue());
      CsvData.append(",");
    }
    // Add the Euler angles data
    for (Index = 0; Index < AnglesDataset3.getSeriesCount(); Index++) {
      CsvData.append(
        AnglesDataset3.getSeries(Index).getY(LastIndex).doubleValue());
      CsvData.append(",");
    }
    // Add the UAV positions data
    for (Index = 0; Index < PositionsDataset1.getSeriesCount(); Index++) {
      CsvData.append(
        PositionsDataset1.getSeries(Index).getY(LastIndex).doubleValue());
      CsvData.append(",");
    }
    for (Index = 0; Index < PositionsDataset2.getSeriesCount(); Index++) {
      CsvData.append(
        PositionsDataset2.getSeries(Index).getY(LastIndex).doubleValue());
      CsvData.append(",");
    }
    for (Index = 0; Index < PositionsDataset3.getSeriesCount(); Index++) {
      CsvData.append(
        PositionsDataset3.getSeries(Index).getY(LastIndex).doubleValue());
      CsvData.append(",");
    }
    // Add the controllers output data
    for (Index = 0; Index < ControlOutputData.getSeriesCount(); Index++) {
      CsvData.append(
```

```
        ControlOutputData.getSeries(Index).getY(LastIndex).doubleValue());
      CsvData.append(",");
    }
    // Add the motor controllers output data
    for (Index=0;Index < MotorControlOutputData.getSeriesCount(); Index++) {
      CsvData.append(
        MotorControlOutputData.getSeries(Index).getY(LastIndex).doubleValue());
      CsvData.append(",");
    }
    // Add the extra data
    for (Index = 0; Index < ExtraData.getSeriesCount(); Index++) {
      CsvData.append(
        ExtraData.getSeries(Index).getY(LastIndex).doubleValue());
      CsvData.append(",");
    }
    CsvData.append("\n");
    LogFile.write(CsvData.toString()); // Write data
    LogFile.flush();
  }
}
```

Fig. E.4 CSV data captured file write routine.

F. DAUGHTER-BOARD FIRMWARE

```
/*************************************************************************/
/* ADC initialization function                                           */
/*************************************************************************/
void
ADC_init (
  void
  )
{
  PORTC &= 0xC0; //Disable pull ups in port C5/4/3/2/1/0
  PORTD &= 0x3F; //Disable pull ups in port D7/6
  ADMUX = 0; //AREF (internal Vref turned off), 10-bit precision, ADC0 act.
  //ADC en., manual triggering, ADC int. en., ADC clock to XTAL/2 = 8 MHz
  ADCSRA = ((1 << ADEN) | (1 << ADIE) | (1 << ADPS0));
  DIDR0 = 0xFF; //Digital inputs disabled on ADC7/6/5/4/3/2/1/0
  ADCSRA |= (1 << ADSC); // Start first conversion
}
```

Fig. F.1 ADC initialization code setting manual triggering, 10-bit conversion precision, and 1 MHz ADC clock.

```
/*************************************************************************/
/* ADC conversion interrupt                                              */
/*************************************************************************/
ISR (ADC_vect)
{
  uint8_t  channelVal;
  uint32_t lowVal;
  // Wait for the conversion complete in case it is set
  while (ADCSRA & (1<< ADSC));
  channelVal = ADMUX & 0xF;
  // Update the ADC sum variable with the high and low bytes of conversion
  lowVal = ADCL;
  adc_sum[channelVal] += (ADCH << 8) | lowVal;
  adcSamples += 1; // Increase the ADC sample counter
  // If number of samples desired has been reached, clear the counter
  if (adcSamples == NUM_ADC_SAMPLES) {
    // Save the sum of the samples in the structure for sending through UART
    motorParameters[channelVal].adcValue = adc_sum[channelVal];
    adc_sum[channelVal] = 0;
    // When the last ADC channel has been reached, finish the ADC sampling
    if ((channelVal + 1) == MAX_MOTORS) {
      adcSamplingTaskStatus = TASK_STATUS_FINISHED;
    }
    // Update the active ADC channel to the next
    ADMUX = (channelVal<(MAX_MOTORS - 1))?(channelVal + 1):(ADMUX & ~(0xF));
    _delay_us (0.5); // Wait 1 ADC clock before the next conversion
    adcSamples = 0; // Clear counter
  }
  ADCSRA |= (1 << ADSC); // Start conversion manually
}
```

Fig. F.2 ADC conversion interrupt showing the storage of the results in the adc_sum array, and the change of the channel reference for conversion.

```
/*************************************************************************/
/* USART initialization function                                         */
/*************************************************************************/
void
USART_init (
  void
  )
{
  UCSR0A = 0; // No parity error
  // Receive interrupt enable, Receiver enable, Transmitter enable, 8-bits
  // to receive/transmit
  UCSR0B = ((1 << RXCIE0) | (1 << RXEN0) | (1 << TXEN0));
  // Asynchronous USART, Parity mode disabled, 8-bits to receive/transmit
  UCSR0C = ((1 << UCSZ01) | (1 << UCSZ00));
  UBRR0 = 1; // UBRRn = 16E6/(16*500000) - 1= 16E6/8E6 - 1 = 2 - 1 = 1
}
```

Fig. F.3 USART initialization function as asynchronous communication, with data reception interrupts enabled and 8 bits to be received/transmitted.

```
/*************************************************************************/
/* GPIO ports initialization function                                    */
/*************************************************************************/
void
GPIO_init (void) {
  //PCINT0:2 are occupied by PB0, PB1 and PB2
  //PCINT3:4 are used for pin voltage level change interrupts
  //PCINT6:7 are occupied by crystal oscillator pins
  //PCINT8:13 are occupied pins by ADC0:6
  //PCINT16:17 are occupied by Rx and Tx UART pins
  //PCINT18:21 and 23 are used for pin voltage level change interrupts
  //Port D pins D7 (ext. data Tx int.), D5, D4, D3 and D2 as inputs
  DDRD &= ~((1 << DDD7)|(1 << DDD5)|(1 << DDD4)|(1 << DDD3)|(1 << DDD2));
  PCICR = 0; // Do not enable interrupts yet until the scheduler takes care
  PCMSK0 = 0; //Do not set these int. pins now, sensor handler does it
  //Interrupt on voltage level change in port D pin D7 and D2 (first motor)
  PCMSK2 = (1 << PCINT23) | (1 << PCINT18);
}
```

Fig. F.4 General-purpose input-output (GPIO) initialization function for Port D pin D3 as input and for voltage level change interrupt.

```
/*************************************************************************/
/* Receive interrupt from USART                                          */
/*************************************************************************/
ISR (USART_RX_vect)
{
  uint8_t AdcQuantity = USART_Receive ();
  TransmitDataASCII (AdcQuantity);
}
```

Fig. F.5 USART receive interrupt executing the receive sequence, and transmitting the data required to the UAV controller.

```
/*************************************************************************/
/* PORT D pin D3 voltage level change interrupt                          */
/*************************************************************************/
ISR (PCINT2_vect) {
  DisableAdcInterrupts ();
  TransmitDataRaw ('4');
  EnableAdcInterrupts ();
}
```

Fig. F.6 Port D pin D3 interrupt executing the transmission of data to the UAV controller.

```
/*************************************************************************/
/* Routine to disable ADC interrupts when doing other task               */
/*************************************************************************/
void DisableAdcInterrupts (void) {
  ADCSRA &= ~(1 << ADIE);
}
```

Fig. F.7 ADC conversion interruption disabling routine to discard ADC data array corruption while transmitting.

```
/*************************************************************************/
/* Enable ADC int. once the task being executed finishes, start ADC conv*/
/*************************************************************************/
void EnableAdcInterrupts (void) {
  ADCSRA |= (1 << ADIE);
  ADCSRA |= (1 << ADSC);
}
```

Fig. F.8 ADC conversion interruption enabling routine for restoring interrupts, once the transmission task has been completed, and start conversion bit setting.

```
/*************************************************************************/
/* This routine takes the number of ADC values to be sent through USART */
/*************************************************************************/
void TransmitDataRaw (uint8_t NumDataToTransmit) {
  uint8_t index, data = NumDataToTransmit;
  DisableAdcInterrupts ();
  // If the data received is a digit (ASCII), then subtract '0' from it
  if ((data >= '0') || (data <= '9')) {
    data -= '0';
    // Send if the digit is not greater than the maximum number of ADCs
    if (data <= MAX_ADC) {
      TransmitPackage (data);
    }
  }
  EnableAdcInterrupts ();
}
```

Fig. F.9 Data transmission routine that shows the disabling of ADC interrupts, the transmission of the data package, and the re-enabling of the ADC interrupts.

```
/*************************************************************************/
/* Timers initialization function                                        */
/*************************************************************************/
void Timer_init (void) {
  // (Timer 1) Normal port operation mode, OC1A and OC1B pins disconnected
  TCCR1A = 0;
  // (2^16)*(0.5us) = 65536*(0.5us) = 32.768ms (time it takes to overflow)
  TCCR1B = (1 << CS11); //16E6/Prescaler = 16E6/8 = 2 MHz = 0.5us period
  TIMSK1 = (1 << TOIE1); // Timer 1 overflow interrupt enable
}
```

Fig. F.10 Timer 1 initialization function, with the period set to 0.5 μs.

```
/*************************************************************************/
/* Routine to set the flag to sample the time for the interrupt pin      */
/*************************************************************************/
void CheckIfSamplingIsRequired (void) {
          uint8_t portPin;
  volatile uint8_t *port = 0;
  LookupSensorIndex (motorSensorActive, port, &portPin);
  if (*port & (1 << portPin))
    UpdatePortPinSampleTime (motorSensorActive);
}
/*************************************************************************/
/* Routine to check if data request has been received from the UAV       */
/*************************************************************************/
void CheckDataRequestStatus (volatile uint8_t *port,
                             uint8_t dataRequestPin) {
  uint8_t newDataRequestPinStatus = *port & (1 << dataRequestPin);
  if (oldDataRequestPinStatus != newDataRequestPinStatus) {
    oldDataRequestPinStatus = newDataRequestPinStatus;
    // Set the flag to send the data to the UAV controller
    isDataRequired = 1;
  }
}
/*************************************************************************/
/* PORT B pins voltage level change interrupt                            */
/*************************************************************************/
ISR (PCINT0_vect) {
  // Prevent to check the pin for speed sampling if data request is active
  if (!isDataRequired)
    CheckIfSamplingIsRequired ();
}
/*************************************************************************/
/* PORT D pins voltage level change interrupt                            */
/*************************************************************************/
ISR (PCINT2_vect) {
  CheckDataRequestStatus (&VOLTAGE_CHANGE_PORTD, DATA_REQUEST_PIN);
  // Prevent to check the pin for speed sampling if the data request is set
  if (!isDataRequired)
    CheckIfSamplingIsRequired ();
}
```

Fig. F.11 Pin voltage level change interrupts routines and the main functions involved.

```
/*************************************************************************/
/* Function that takes care of enabling/disabling ADC and GPIO pin      */
/* voltage level change interrupts in a scheduler fashion to not have   */
/* overlaps                                                              */
/*************************************************************************/
void
SchedulerHandler (
  void
  )
{
  // If ADC sampling task has completed, set the task in halt, set the speed
  // sampling task in running, disable ADC conversion interrupts, enable pin
  // level change interrupts and clear the stage for the motor sensor active
  if (adcSamplingTaskStatus == TASK_STATUS_FINISHED) {
    adcSamplingTaskStatus = TASK_STATUS_HALT;
    speedSamplingTaskStatus = TASK_STATUS_RUNNING;
    time_capture_stage[motorSensorActive] = AFTER_TIMER_OVERFLOW;
    DisableAdcInterrupts (); EnablePinChangeInterrupts ();
  }
  // If the speed sampling has completed, set the task in halt, set the ADC
  // conversion task in running, disable pin level change interrupts, and
  // enable ADC conversion interrupts
  if (speedSamplingTaskStatus == TASK_STATUS_FINISHED) {
    speedSamplingTaskStatus = TASK_STATUS_HALT;
    adcSamplingTaskStatus = TASK_STATUS_RUNNING;
    DisablePinChangeInterrupts (); EnableAdcInterrupts ();
  }
}
```

Fig. F.12 ADC conversion and pin voltage level change interrupt scheduler handler routine.

Bibliography

[Adamo-17] F. Adamo, G. Andria, A. D. Nisio, C. G. C. Carducci, A. M. L. Lanzolla, and G. Mattencini, "Development and characterization of a measurement instrumentation system for UAV components testing," in *Int. Workshop on Metrology for AeroSpace*, Padua, Italy, Jun. 2017, pp. 355-359.

[ADT-16a] ArduPilot Dev Team. (2016), *Connect ESCs and Motors (rev. 2019)* [Online]. Available: http://ardupilot.org/copter/docs/connect-escs-and-motors.html.

[ADT-16b] ArduPilot Dev Team (2016), *PX4FLOW Optical Flow Camera Board Overview (rev. 2019)* [Online]. Available: http://ardupilot.org/copter/docs/common-px4flow-overview.html.

[ADT-18a] ArduPilot Dev Team (2018), *Archived: APM 2.5 and 2.6 Overview (rev. 2019)* [Online]. Available: http://ardupilot.org/rover/docs/common-apm25-and-26-overview.html#common-apm25-and-26-overview.

[ADT-18b] ArduPilot Dev Team (2018), *Community: ArduPilot (rev. 2019)* [Online]. Available: http://ardupilot.org/ardupilot/.

[ADT-19a] ArduPilot Dev Team (2019), *Advanced Compass Setup (rev. 2019)* [Online]. Available: http://ardupilot.org/copter/docs/common-compass-setup-advanced.html.

[ADT-19b] ArduPilot Dev Team (2019), *ST VL53L0X/VL53L1X Lidar (rev 2019)* [Online]. Available: http://ardupilot.org/copter/docs/common-vl53l0x-lidar.html.

[AIAA-18] American Institute of Aeronautics and Astronautics (2018), *Leonardo Da Vinci (1452-1519) (rev. 2018)* [Online]. Available: https://www.aiaa.org/SecondaryTwoColumn.aspx?id=15129.

[Alkurawy-18] L. E. J. Alkurawy, and N. Khamas, "Model predictive control for DC motors," in *Int. Scientific Conference of Engineering Sciences*, Diyala, Iraq, Jan. 2018, pp. 56-61.

[Allegro-17] *ACS712 Fully Integrated, Hall Effect-Based Linear Current Sensor IC with 2.1 kVRMS Isolation and a Low-Resistance Current Conductor*, Allegro MicroSystems, LLC, Worcester, MA, 2017.

[Arellano-Muro-13] C. A. Arellano-Muro, L. F. Luque-Vega, B. Castillo-Toledo and A. G. Loukianov, "Backstepping control with sliding mode estimation for hexacopter," in *Int. Conf. on Electrical Engineering, Computing and Automatic Control*, Mexico City, Mexico, Sep. 2013, pp. 31-36.

[Atmel-16] *ATmega328/P Datasheet Complete*, Atmel Corporation, San Jose, CA, 2016.

[Austin-10] R. Austin, *Unmanned Aircraft Systems: UAVS Design, Development and Deployment*, Chichester, West Sussex: WILEY, 2010.

[Baránek-12] R. Baránek, and F. Šolc, "Modelling and control of a hexa-copter," in *Int. Carpathian Control Conference*, High Tatras, Slovakia, May. 2012, pp. 19-23.

[Bouabdallah-05] S. Bouabdallah, and R. Siegwart, "Backstepping and sliding-mode techniques applied to an indoor micro quadrotor," in *Int. Conf. on Robotics and Automation*, Barcelona, Spain, Jan. 2005, pp. 2247-2252.

[CFA-18] Canadian Fuels Association (2018), *Drone technology makes Shell's Scotford Refinery even safer (rev. 2018)* [Online]. Available: http://www.canadianfuels.ca/Blog/January-en/Drone-technology-makes-Shell-s-Scotford-Refinery-even-safer/.

[Chao-13] H. Chao, Y. Gu, and M. Napolitano, "A survey of optical flow techniques for UAV navigation applications," in *Int. Conf. on Unmanned Aircraft Systems*, Atlanta, GA, May 2013, pp. 710-716.

[Chen-14] F. Chen, R. Jiang, K. Zhang, B. Jiang, and G. Tao, "Robust Backstepping Sliding-Mode Control and Observer-Based Fault Estimation for a Quadrotor UAV," *IEEE Transactions on Industrial Electronics*, vol. 63, no. 8, pp. 5044-5056, Aug. 2016.

[Cleanflight-18] Cleanflight (2018), *Cleanflight (rev. 2018)* [Online]. Available: https://github.com/cleanflight/cleanflight.

[Davis-17] E. Davis, and P. E. I. Pounds, "Direct sensing of thrust and velocity for a quadrotor rotor Array," *IEEE Robotics and Automation Letters*, vol. 2, no. 3, pp. 1360-1366, Jul. 2017.

[Derafa-12] L. Derafa, A. Benallegue, and L. Fridman, "Super twisting control algorithm for the attitude tracking of a four rotors UAV," *J. of the Franklin Institute*, vol. 349, pp. 685-699, Mar. 2012.

[Dronecode-18] Dronecode (2018), *PX4FLOW Smart Camera (rev. 2018)* [Online]. Available: https://docs.px4.io/en/sensor/px4flow.html.

[Duc-15] M. N. Duc, T. N. Trong, and Y. S. Xuan, "The quadrotor MAV system using PID control," in *Int. Conf. on Mechatronics and Automation*, Beijing, China, Aug. 2015, pp. 506-510.

[Ehsani-99] M. Ehsani, M. Masten, and I. Panahi, "Stiff system control: a new concept in real-time control," in *American Control Conference*, San Diego, CA, Jun. 1999, pp. 2125-2136.

[EMAX-18] EMAX (2018), *MT2204 SPECIFICATIONS (rev. 2018)* [Online]. Available: https://www.emaxmodel.com/emax-multicopter-motor-mt2204-kv2300.html.

[EMLID-18] EMLID (2018), *NAVIO2 (rev. 2018)* [Online]. Available: https://emlid.com/navio/.

[Erle-Robotics-18a] Erle Robotics (2018), *Erle-Brain 3 (rev. 2018)* [Online]. Available: http://docs.erlerobotics.com/brains/erle-brain-3.

[Erle-Robotics-18b] Erle Robotics (2018), *PXFMINI (rev. 2018)* [Online]. Available: https://erlerobotics.com/blog/product/pxfmini/.

[Everett-11] J. E. Everett, "The exponential weighted moving average applied to the control and monitoring of varying sample sizes," in *Int. Conf. on Computational Methods and Experimental Measurements*, New Forest, United Kingdom, Jun. 2011, pp. 3-13.

[Fan-17] Y. Fan, Y. Cao, and Y. Zhao, "Design of the nonlinear controller for a quadrotor trajectory tracking," in *Chinese Control And Decision Conference*, Chongqing, China, Jul. 2017, pp. 2162-2167.

[Felismina-17] R. Felismina, M. Silva, A. Mateus, and C. Malca, "Study on the aerodynamic behavior of a UAV with an applied seeder for agricultural practices," *American Institute of Physics Conf. Proc.*, vol. 1836, no. 1, pp. 1-14, Jun. 2017.

[FlytBase-18] FlytBase (2018), *Operating System for Drones (rev. 2018)* [Online]. Available: https://flytbase.com/flytos/.

[GC-18] GRABCAD COMMUNITY (2015), *ZMR250 Quadcopter Frame (rev. 2015)* [Online]. Available: https://grabcad.com/library/zmr250-quadcopter-frame-1.

[Giusti-15] A. Giusti, J. Guzzi, and D. C. Cireşan, "A Machine Learning Approach to Visual Perception of Forest Trails for Mobile Robots," *IEEE Robotics and Automation Letters*, vol. 1, no. 2, pp. 661-667, 2015.

[González-Jiménez-20] L. E. González-Jimenez (2020), *Reference Tracking in Stabilize Flight Mode for a Quadrotor Using Velocity Sensors in the Actuators* [Online]. Available: https://youtu.be/RdDv3WunO1Y.

[González-Hernández-12] I. González-Hernández, "Attitude stabilization of a Quad-rotor UAV based on rotor speed sensing with accelerometer data estimation via Kalman filtering," in *31st Chinese Control Conference*, Hefei, China, Jul. 2012, pp. 5123-5128.

[Greitzer-03] E. M. Greitzer, Z. S. Spakovszky, and I. A. Waitz (2013), *11.7 Performance of propellers (rev. 2003)* [Online]. Available: https://web.mit.edu/16.unified/www/FALL/thermodynamics/notes/node86.html.

[Grossman-12] S. I. Grossman and J. J. Flores Godoy, *Álgebra Lineal*, México, D.F.: McGraw-Hill, 2012.

[Hafifi-Zulkipli-16] A. Hafifi-Zulkipli, T. R. F. Hanim-Hashim, and A. Baseri-Huddin, "Characterization of DC brushless motor for an efficient multicopter design," in *Int. Conf. on Advanced Electrical, Electronic and System Engineering*, Putrajaya, Malaysia, Nov. 16, pp. 586-590.

[Honegger-13] D. Honegger, L. Meier, P. Tanskanen and M. Pollefeys, "An open source and open hardware embedded metric optical flow CMOS camera for indoor and outdoor applications," in *Int. Conf. on Robotics and Automation*, Karlsruhe, Germany, May 2013, 1736-1741.

[IntelinAir-18] IntelinAir (2018), *SafeSmart RD-100 UAV (rev. 2018)* [Online]. Available: http://www.intelinairflight.com/wp-content/uploads/2015/09/150917-RD-100.pdf.

[Jun-Li-11] J. Li and Y. Li, "Dynamic analysis and PID control for a quadrotor," in *IEEE Int. Conf. on Mechatronics and Automation*, Beijing, China, Aug. 2011, pp. 573-578.

[Keeping-13] S. Keeping (2013), *An introduction to Brushless DC Motor (rev. 2013)* [Online]. Available: https://www.digikey.com/en/articles/techzone/2013/mar/an-introduction-to-brushless-dc-motor-control.

[Kester-06] W. Kester, "ADC Input Noise: The Good, The Bad, and The Ugly. Is No Noise Good Noise?," *Analog Dialogue*, vol. 40, no. 1, pp. 13-17, 2006.

[Kumpanya-15] D. Kumpanya, S. Thaiparnat, and D. Puangdownreong, "Parameter identification of BLDC motor model via metaheuristic optimization techniques," *Procedia Manufacturing*, vol. 4, pp. 322-327, 2015.

[Leishman-06] G. J. Leishman, *Principles of Helicopter Aerodynamics*, New York: Cambridge University Press, 2006.

[Lemos-07] R. Lemos (2007), *The helicopter: a hundred years of hovering (rev. 2007)* [Online]. Available: https://www.wired.com/2007/12/gallery-helicopter/.

[Levant-03] A. Levant, "Higher-order sliding modes, differentiation and output-feedback control," *Int. J. of Control*, vol. 76, no. 9-10, pp. 924-941, Sep. 2003.

[Levant-98] A. Levant, "Robust exact differentiation via sliding mode technique," *Automatica*, vol. 34, no. 3, pp. 379-384, 1998.

[LibrePilot-18a] LibrePilot (2018), *CopterControl / CC3D / Atom Hardware Setup (rev. 2018)* [Online]. Available: https://opwiki.readthedocs.io/en/latest/user_manual/cc3d/cc3d.html#id2.

[LibrePilot-18b] LibrePilot (2018), *Main Features (rev. 2018)* [Online]. Available: https://www.librepilot.org/site/index.html.

[Luque-Vega-14] L. F. Luque-Vega, "Nonlinear Control for Multicopters," Ph.D. dissertation, CINVESTAV del IPN Unidad Guadalajara, Guadalajara, 2014.

[Lyons-11] R. G. Lyons, *Understanding digital signal processing*, Boston, MA: Prentice Hall, 2011.

[MacAusland-14] R. MacAusland (2014), *MATH 420: Advanced Topics in Linear Algebra - The Moore-Penrose Inverse and Least Squares (rev. 2014)* [Online]. Available: http://buzzard.ups.edu/courses/2014spring/420projects/math420-UPS-spring-2014-macausland-pseudo-inverse.pdf.

[Mayorga-Macías-20] W. A. Mayorga-Macías (2020), *UAV individual controllers tests* [Online]. Available: https://drive.google.com/drive/folders/1Pd_bB9dhAwmdsawSn9N62-ZI9WRW4b9N?usp=sharing.

[Merheb-17] A. R. Merheb, H. Noura, and F. Bateman, "Emergency control of AR drone quadrotor UAV suffering a total loss of one rotor," *IEEE/ASME Trans. on Mechatronics*, vol. 22, no. 2, pp. 961-971, Apr. 2017.

[Milosavljević-10] Č. Milosavljević, B. Peruničić, and B. Veselić, "A new real differentiator with low-pass filter in control loop," *Electronics*, vol. 14, no. 2, pp. 27-32, 2010.

[Niekerk-15] D. V. Niekerk, M. Case, and D. Nicolae, "Brushless direct current motor efficiency characterization," in *Intl. Aegean Conference on Electrical Machines & Power Electronics*, Side, Turkey, Sep. 2015, pp. 226-231.

[Nonami-10] K. Nonami, F. Kendoul, S. Suzuki, W. Wang, and D. Nakazawa, *Autonomous Flying Robots: Unmanned Aerial Vehicles and Micro Aerial Vehicles*, Tokyo, Japan: Springer, 2010.

[O'Donnell-17] S. O'Donnell (2017), *A short history of Unmanned Aerial Vehicles (rev. 2017)* [Online]. Available: https://consortiq.com/media-centre/blog/short-history-unmanned-aerial-vehicles-uavs.

[Ogata-96] K. Ogata, *Sistemas de Control en Tiempo Discreto*, Edo. de México, México: Prentice Hall, 1996.

[PX4DT-18] PX4 Dev Team (2018), *Pixhawk 1 Flight Controller (rev. 2018)*. [Online]. Available: https://docs.px4.io/en/flight_controller/pixhawk.html.

[Qian-16] M. Qian and Z. Gao, "Adaptive fault-tolerant control design for UAV with multiple actuator faults," in *Chinese Guidance, Navigation and Control Conf.*, Nanjing, China, Aug. 2016, pp. 814-819.

[Qualcomm-18] Qualcomm (2018), *Qualcomm Flight Board Specifications (rev. 2018)* [Online]. Available: https://developer.qualcomm.com/hardware/qualcomm-flight/board-specs.

[RMRC-18] RMRC (2018), *RMRC Seriously Dodo Flight Controller - Rev 3b (rev. 2018)* [Online]. Available: https://www.readymaderc.com/products/details/rmrc-seriously-dodo-flight-controller-rev-3b.

[ROS-18] ROS (2018), *Core components (rev. 2018)* [Online]. Available: http://www.ros.org/core-components/.

[RPAV-03] Remote Piloted Aerial Vehicles (2013), *Remote Piloted Aerial Vehicles: An Anthology (rev. 2003)* [Online]. Available: http://www.ctie.monash.edu.au/hargrave/rpav_home.html.

[Sahputro-17] S. D. Sahputro, F. Fadilah, N. A. Wicaksono, and F. Yusivar, "Design and implementation of adaptive PID," in *Int. Conf. on Quality in Research*, Nusa Dua, Indonesia, Jul. 2017, pp. 179-183.

[Santos-15] M. F. Santos, L M. Honório, E. B. Costa, E. J. Oliveira, and J. P. Portella-Guedes-Visconti, "Active fault-tolerant control applied to a hexacopter under propulsion system failures," in *Int. Conf. on System Theory, Control and Computing*, Cheile Gradistei, Romania, Oct. 2015, pp. 447-453.

[Stamp-13] J. Stamp (2013), *WORLD WAR I: 100 YEARS LATER (rev. 2013)* [Online]. Available: https://www.smithsonianmag.com/arts-culture/unmanned-drones-have-been-around-since-world-war-i-16055939/.

[Strickland-18] E. Strickland (2018), *Drone delivery becomes a reality in remote Pacific islands (rev. 2018)* [Online]. Available: https://spectrum.ieee.org/the-human-os/biomedical/devices/drone-delivery-becomes-a-reality-in-remote-pacific-islands.

[Surya-16] Y. Surya and E. Pitowarno, "Robust adaptive proportional derivative-active force control for unmanned hexacopter," in *Int. Electronics Symp.*, Denpasar, Indonesia, Sep. 2016, pp. 1-6.

[UM-99] University of Michigan (2017), *Introduction: PID controller design (rev. 2017)* [Online]. Available: http://ctms.engin.umich.edu/CTMS/index.php?example=Introduction&section=ControlPID.

[Utkin-99] V. Utkin, J. Guldner and J. Shi, *Sliding Mode Control in Electromechanical Systems*, Philadelphia, PA: Taylor & Francis, 1999.

[Wang-09] L. Wang and Q. p. Wang, "The feedback linearization based on backstepping technique," in *Int. Conf. on Intelligent Computing and Intelligent Systems*, Shanghai, China, Dec. 2009, pp. 282-286.

[Zulu-14] A. Zulu and S. John, "A review of control algorithms for autonomous quadrotors," *Open Journal of Applied Sciences*, vol. 4, no. 14, pp. 547-556, Dec. 2014.

Author Index

Subject Index

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

Y