REPUBLIQUE ALGERIENNE DEMOCRATIQUE ET POPULAIRE
الجمهورية الجزائرية الديمقراطية الشعبية

MINISTRY OF HIGHER EDUCATION AND SCIENTIFIC RESEARCH

HIGHER SCHOOL IN APPLIED SCIENCES
--T L E M C E N--

وزارة التعليم العالي والبحث العلمي

المدرسة العليا في العلوم التطبيقية
-تلمسان-

Mémoire de fin d'étude

Pour l'obtention du diplôme d'Ingénieur

Filière : Automatique
Spécialité : Automatique

Présenté par : ZIANI-KERARTI AbdelKarim

Thème

Quadcopter Control Using Reinforcement Learning

Soutenu publiquement, le 12 / 07 / 2021 , devant le jury composé de :

M. CHERKI Brahim	Professeur	ESSA. Tlemcen	Président
M. MOKHTARI Rida	MCA	ESSA. Tlemcen	Directeur de mémoire
M. ARICHI Fayssal	MCB	ESSA. Tlemcen	Co- Directeur de mémoire
Mme. CHOUKCHOU-BRAHAM Amal	Professeur	UNIV. Tlemcen	Examinateur 1
M. BENSALAH Choukri	MCB	UNIV. Tlemcen	Examinateur 2

Année universitaire : 2020 /2021

ACKNOWLEDGMENTS

First and foremost, I would like to praise and thank God, the almighty, ho has granted countless blessing, and letting me live this amazing experience full of passion and enthusiasm, so that I have been finally able to accomplish the thesis. I would also express my deep and sincere gratitude to my family for their unconditional and unparalleled love, help and support. I am forever indebted to my parents for giving me thee opportunities and experiences that have made me who I am. They selflessly encouraged me to explore new directions in life and seek my own destiny. This journey would not have been possible without them.

Most profound thanks are addressed to Pr. CHERKI Brahim for accepting to be the president of the examiners, and also for his immense knowledge, plentiful experience in several different fields, his working methodology, his enthusiasm and his teaching quality that have encouraged me all the time. Special thanks to Pr. CHOUKCHOU-BRAHAM Amal and BEN-SALAH Choukri for the particular interest they have taken in my work and the big pleasure they gave me by accepting to be part of the examiners. Their advises and remarks have been very precious to me.

My sincere thanks are extended to my advisors Dr. MOKHTARI Rida and ARICHI Fayssal for their supervision and technical support.

Apart from the effort of me, the success of this thesis depends on the encouragements, guidance, and contribution of Dr. RAHMOUN MRABET Rassia, from the School of Applied

Sciences (ESSA) Tlemcen, to Jacopo Panerati and Erwin Couman for providing the amazing Github repository that i used to make this project possible.

Finally, it is a pleasure to thank all my friends especially Mohammed, Omar, Fakhro, Reda and Ahmed for the wonderful times we shared and their infinite support.

Contents

List of Figures

GENERAL INTRODUCTION

Aerial robots has grown in popularity over the last decade as a result of technological advancements and several potential uses. Unmanned Aerial Vehicles known as "rotorcraft", are self-flying aircraft with one or more rotating wings or rotors that produce lift and propulsion, and they are capable of following trajectories, navigating in space, employing visual navigation, taking off, as well as performing quasi-stationary and low-altitude missions with great maneuverability. They have access to a wind range of military and civilian applications, including aerial surveillance, intelligence tactical reconnaissance, exploration of unknown environments, warefare and transportation.

Autopilots for mini-UAVs provide a number of theoretical and technical hurdles, as well as technological challenges. Indeed, the limited the payload of miniature vehicles imposes sever limitations on the navigation sensors and on-board electronics that can be used. Furthermore, due to their complex dynamics, non-linearities, and high degrees of coupling between the different degrees of freedom, as well as the dynamics of the actuators, the synthesis of control laws becomes problematic. From the perspective of a control system, most control models involve a trade-off between performance and flying mode, as well as the complexity of the control law. Machine learning in particular Reinforcement Learning, has reduced the numbers of challenges faced by aerial robotics in general beside enhancing the capabilities and opening the door to different sectors. Reinforcement Learning has evolved a long way with the enhancements from deep learning. Recent research efforts into combining deep learning with Reinforcement Learning have led to the development of some very powerful Deep Reinforcement Learning systems, algorithms, and agents which have already achieved some extraordinary accomplishments. Not even such systems surpassed the capabilities of most the classical control strategies, Reinforcement Learning agents have also started outperforming the best of human intelligence at tasks which were believed to require extreme human intelligence, creativity, and planning tasks. Some Reinforcement Learning agents consistently beating the best human players at complex games such as Google's AlphaGO beating the best human player in GO.

In Reinforcement learning, an agent is given a reward for every action it takes in an environment, with the objective to minimize the reward over time. Using Reinforcement Learning, it is possible to develop optimal control policies for Unmanned Aerial Vehicles without making any assumptions about the aircraft dynamics. Recent works in robotics research community has

shown Reinforcement Learning to be effective for Unmanned Aerial Vehicles autopilots, providing adequate path tracking, real-time monitoring, data collection and processing, monitoring in smart cities, military, agriculture and mining.

In this thesis, we talked about Unmanned Aerial Vehicles history, from early designs to more modern intelligent aircraft, then we developed a mathematical model for the quadcopter using the Newton-Euler formalism, then a classical backstepping control strategy for attitude control. Finally, we study in depth the accuracy and precision of attitude control provided by intelligent flight controllers trained using Reinforcement Learning. We developed a novel training environment using the OpenAI GYM library with the use of the PyBullet high-fidelity physics simulator for the agent to learn and train.

This thesis is divided into five chapters, each of which is briefly described in the paragraph below:

- **Chapter 1**: presents a history of Unmanned Aerial Vehicles along with a classification of aerial systems according to three main families: fixed-wings, flapping-wing and rotary-wings.
- **Chaapter 2**: covers a brief description of the state-of-the-art commercial flight controllers currently in use.
- **Chapter 3**: covers in the first part a mathematical description of the dynamical model of the quadcopter using the Newton-Euler formalism. The second part of the chapter is dedicated to the design of a classical control strategy using the backstepping control procedure for attitude control.
- **Chapter 4**: An introduction to Reinforcement Learning paradigm and the use of Markovian decision process environments and Dynamic Programming for solving the optimization criterion. Later on we gave a description of the Reinforcement Learning from a control system strategy and the similarities between the two promising field. And finally, we spoke about Value Iteration and Policy Iteration as the two main approaches for solving the Reinforcement Learning probelm.
- **Chapter 5**: covers the quadcopter control using Reinforcement Learning and its main advantages over the classical control schema. After that we familiarized ourselves with the git version control and the docker virtualization techniques, the PyBullet physics engine used alongside OpenAI GYM library for creating the environment for the Reinforcement Learning agent to train.

CHAPTER

1

INTRODUCTION

The interest toward Aerial Robotics and unmanned aerial vehicles (UAVs) such as drones is daily growing these recent decades, not only within research communities, but also among industrial companies, military and also the open public such as hobbyists. This emerging technology showed a huge potential in various sectors to provide a wide range of applications and services to improve human life and the quality of service. The thriving focus of research in UAV's assistance paradigm as a fundamental concept resulted in a constant improvement and an increasing usage in a wide range of civil and domestic applications such as: photography, surveillance, Internet of Things (IoT), environment monitoring, search and rescue, product delivery and agriculture, thanks to their decreased weight, reduced size, low cost and their increasing functionalities.

1.1 History

1.1.1 The First Projects

Although, originally built for military purposes, the drones have experienced an exponential progress and advancements and made a break to consumer electronics but the original idea of a "flying machine" preoccupied man since the beginning of time. It was documented that the major breakthrough contribution and early design for a flying machine occurred during the era of Pythagoras. By applying a series of geometric notions and observations to the study of structures, links and joints, Archytas the Tarantine (also referred to as Leonardo Da Vinci of the ancient world) created the first UAV of all times in 425 B.C by building a mechanical steam-powered flying pigeon (in Greek περιστέρι) that could fly by moving its wings. It was called that way because of its structure that reassembles a bird. This invention also known as "the first autonomous volatile machine of antiquity" was a highly advanced invention for that time and it was one of the first studies into how birds fly. The Flying Pigeon was lightweight because it was built from a hollow cylindrical shape piece of wood with wings projected out

to either side, and smaller wings to the rear. The rear of the flying pigeon had an opening that led to the internal bladder and it was connected to a heated, airtight boiler, this boiler created more steam powering the pigeon. The pressure of the steam eventually exceeded the mechanical resistance of the connection, and the flying pigeon took flight. It is alleged that it flew about 200 meters before falling to the ground once all energy was used.

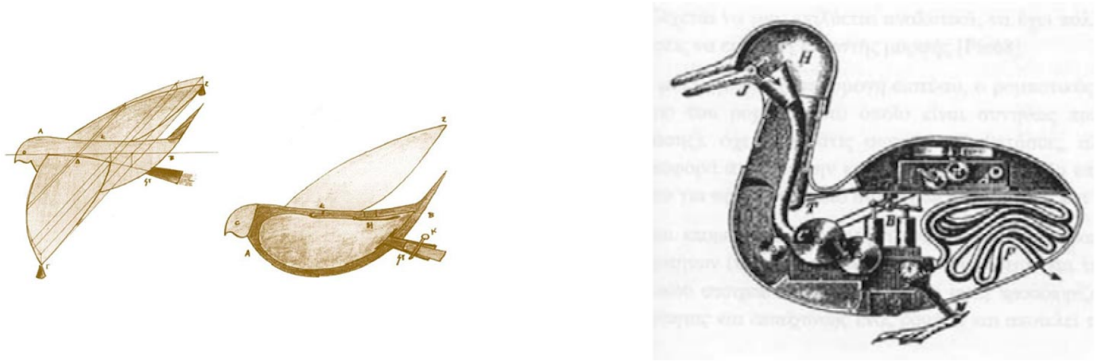


Figure 1.1: Flying Pigeon

Around that same era, at another part of the ancient world, in China at about 400 B.C the Chinese were the first to document the idea of a vertical flight aircraft known as Bamboo flying toy. This Bamboo-copter is spun by rolling a stick attached to a rotor. The spinning creates lift and the toy flies when released.

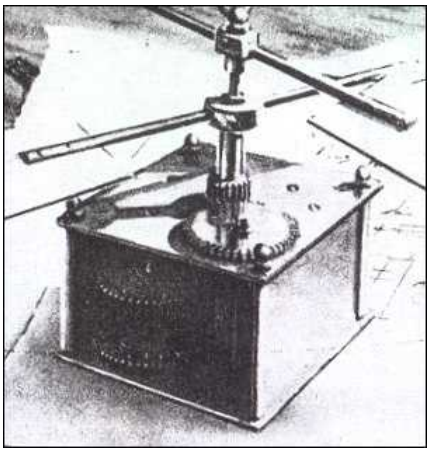
Figure 1.2: Bamboo-copter

In 1483, The Italian polymath Leonardo Da Vinci designed an aircraft called "the Aerial Screw" or "Air Gyroscope", it was one of several aerial machines designed by Leonardo including an early Parachute, an Ornithopter and a Hang Glider. This aircraft comprises a large structure built on a solid circular platform with a central vertical pole supported by three diagonal members meeting at a small circular plate about half way up the pole. The structure had a 5 meter diameter and the idea was to make the shaft turn and if enough force were applied, the machine could spun and fly. The inner edge of the sail winds clockwise around the pole, while the outer edge of the sail is connected by ropes or wires to a ring that rotates around the lower platform. Some experts consider that the modern design of the today's helicopter is heavily inspired by Da Vinci's Aerial Screw aircraft.

Further, Da Vinci in 1508 designed a flying mechanical bird that reassembles Archytas Flying Pigeon. It could flap its wings by the help of a double crank mechanism as it descended along a cable.

Figure 1.3: Air Gyroscope

Two additional designs directly inspired by the Chinese Bamboo-copter were documented in 1754 and 1783 respectively. The very first self-propelled model of a lifting airscrew constructed and flown by the the "Father of Russian Science" Mikhail Lomonosov. He demonstrated his invention called "Aerodynamic" to be used for the purpose of depressing the air by means of wings rotated horizontally in the opposite directions by the agency of a spring of the type used in clocks in order to lift the machine into the air. In July 1754, he presented his model to the Russian Academy of Science and it appeared that during the course of his presentation, the model was not actually in free flight but was suspended from a string. The second design was credited to Launoy and Bienvenue whose model consisted of a counter rotating set of turkey feathers.



(a) Coaxial rotor, Lomonosov

(b) Contra-rotating propeller, Launoy and Bienvenue

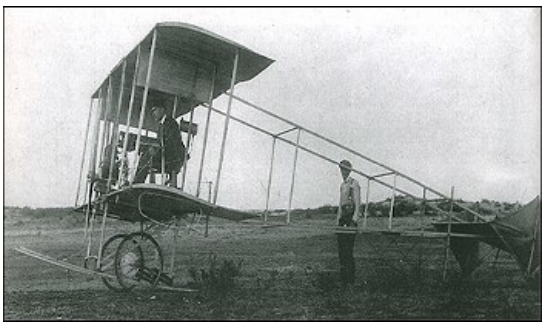
If practical results were not forthcoming, there was one memorable advance achieved during 1860 by the French pioneer Ponton d'Amecourt who flew a small steam-driven helicopter and took out French and British patents on it. It was at that time the term "helicopteres" was first coined, based on the greek word "ελικόπτερο" that is composed of two words, "ελικας" referring

to something that spins (spiral) and "πτερον" that means feather (like a bird feather) or wing (like airplane wing).

Figure 1.5: steam-driver helicopter, d'Amecourt

The major breakthrough in helicopter history of modern times was the Sikorsky S-1 aircraft. Designed by the Russian-American aviation pioneer Igor Ivanovich Sikorsky in 1910, the S-1 was the first fixed wing, non-piloted coaxial helicopter powered by a 15 hp (11 kW) Anzani three-cylinder, air-cooled engine. The first successful flight attempt by Sikorsky was in early May of that same year, during a take-off on a windy day, the helicopter briefly became airborne due mostly to its well designed headwind. Further attempts were made but they were less successful and Sikorsky decided to disassemble the S-1 and save the main wing section to a much improved design. In June 1910, Sikorsky constructed his second fixed wing helicopter by using the same main wing section from the S-1 and a more powerful 25 hp (19kW) Anzani engine in a tractor configuration. Later that same month, several successful flights took place before the S-2 got completely destroyed when Sikorsky inadvertently stalled the underpowered aircraft at an altitude of 70 feet.

(a) Sikorsky S-1 aircraft



(b) Sikorsky S-2 aircraft

1.1.2 Pre-World War Designs

Before World War I, the earliest record use of unmanned aerial vehicles for warefare occurred in 1849, serving as a balloon carrier, Austrian forces besieging Venice attempted to float some 200 incendiary balloons each carrying a 24 to 30 pound of explosives . The balloons were mainly launched from land, however some were also launched from the SMS Volcano Austrian ship.

The Austrian military also used smaller pilot balloons to determine the correct fuse settings and hit the target precisely. However, these balloons don't generally meet the modern definitions of unmanned aerial vehicles. Therefore, it is accepted that the first pilotless aircraft used in warfighting was suggested from Archibald Montgomery Low. He has been called "the father of radio guidance systems" due to his pioneering work on planes, torpedoes boats and guided rockets. His expertise in television and radio technology was being used to develop one of the earliest remotely controlled pilotless aircraft to attack the Zeppelins. This led to a remarkable succession of British drone weapons in 1917 and 1918. Several aeroplane manufacturing companies in the United States and Europe especially in the United Kingdom such as Sopwith Aviation and its contractor Rushton Proctor, de Havilland and the Royal Aircraft Factory, began constructing and producing remotely controlled aircrafts that involved Low's ingenious radio control system that was firstly secretly developed at the British Royal Corps for the British Royal Naval Air Service and Royal Air Force. Soon after, the Hewitt-Sperry Automatic Airplane otherwise known as the "flying bomb" made its first flight, emphasizing the concept of an unmanned aircraft. The control was achieved using gyroscopes developed by the Gyroscope company and they were intended to be used as aerial torpedoes as an early version of today's cruise missiles.

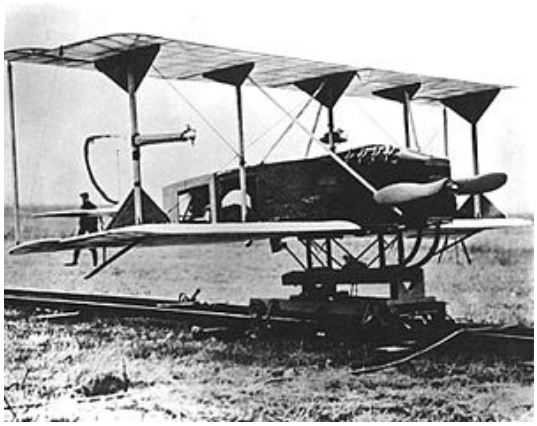


Figure 1.7: Hewitt-Sperry automatic airplane

Early US efforts in late 1917, came with the Kettering Bug experimental unmanned aerial torpedo named after the American inventor and Engineer Charles Kettering. It was a project commissioned by the US army for creating a flying bomb capable of striking ground targets up to 121 kilometers from its launch point, while traveling at speed of 80 kph (kilometers per hour). The Bug was designed with a small on-board gyroscope like the one used in Hewitt-Sperry airplane. Its main goal is to guide the aircraft to its destination at approximately 190 kph, and the control was maintained by using a pneumatic/vacuum system, electric system and an aneroid barometer/altimeter. To hit the ground target precisely, technicians had to plot the craft's trajectory and predict approximately how many engine revolutions were necessary to reach the target using a mechanical system that can track the distance the aircraft flew. While the Bug's revolutionary technology was successful, it was not in time to flight in the war, which ended before it could be fully developed and deployed.

1.1.3 Interwar period

After the World War I, three standard American Army Fighter aircraft were converted to drones and they were the early cruise missile monoplane aircrafts that flown under autopilot. They

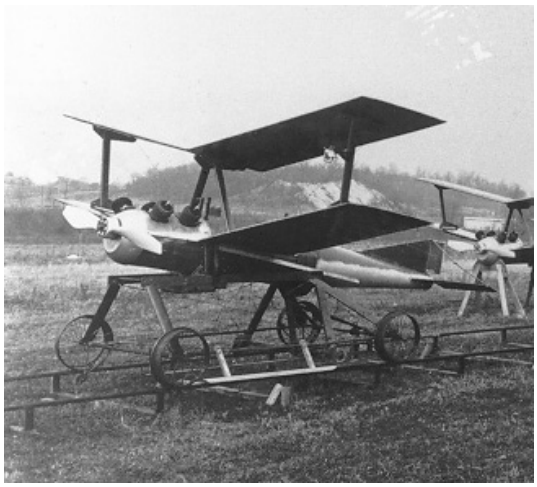


Figure 1.8: Kettering Bug airplane

were tested between 1927 and 1929 by the American Royal Navy. Around that same time, the development of radio controlled pilotless aircraft was taking place, notably the invention of the Fairey Queen floatplane, the DH.82B Queen Bee and the Airspeed Queen Wasp by the British, the first infrared-sensitive (night vision) electronic television camera for anti-aircraft defense, and remotely guided aircrafts that utilizes prototypes of a camera by the Hungarian scientist Kálmán Tihanyi. His solutions were so influential to the point that many aircraft manufacturing companies in Britain and the US still used his solutions even half a century later, until the mid 1980s.

After the noteworthy success of the radio controlled aircrafts, a large-scale production of purpose-build drones began to see the lights around the second World War. The Radioplane OQ-2 was the first mass-produced UAV in the US manufactured by the Radioplane Company used for drone targeting. Around 1937, the US Navy began experimenting with the Curtiss N2C-2 drone which later was adopted by the US Army Air Forces (USAAF) in 1939 resulting in hundred versions of Culver PQ-8, Culver PQ-14 Cadet and an improved modified version B-17 1flying Fortress and B-24 Librator which were very large aerial torpedoes used as heavy bombers in Aphrodite and Anvil operations on a small scale during World War II.

1.2 The New Era in UAVs

The development of aerial drones is growing rapidly and its increasing interest is attracting not only the military but also the open public. This innovative and game-changing technology has the potential to transform commercial industries and open limitless future opportunities in the field of aerial robotics. Drones are already breaking barriers in the way companies do business. Huge corporations like Amazon and Google are testing ways to deliver packages and product with drones, Facebook also is using drones to provide internet connections in remote location. Below, are some of the many ways UAVs are being used to maximize productivity and do various things that could never be possible before:

- **Aerial Photography**: Drones are now being use by both professional film makers and hobbyists to capture footage that would otherwise require expensive helicopters and cranes. Furthermore, these autonomous flying devices are also used by journalists for

collecting footage and information in live broadcast eliminating the need of putting human life in danger especially in the coverage of sensitive dangerous areas like warzones.

- **Shipping and Delivery**: Major techology companies like Amazon, UPS and DHL are in favor of drone delivery. Thanks to their reduced size and decreasing lightweight, they could save a lot of manpower and shift unnecessary road traffic to the sky. Besides, they can also be used over small distances for the delivery of food, letters, medicines and beverages.
- **Geographic Mapping**: Modern high-technology drones can acquire very high-resolution data and download imagery in difficult to reach locations like coastlines, rain-forests, mountaintops and islands. They are also being used to create very-detailed 3D maps and contribute to crowd sourced mapping applications.
- **Disaster Management**: Drones are used in rescue and firefighting missions to gather information and navigate debris in natural disaster, help and search for survivors saving the need to spend resources on manned helicopters and radars. Moreover, telecommunication, water and electricity companies are also adopting drone technology to assess their utilities integrity and infrastructural damage.
- **Precision Agriculture**: For monitoring the crops, farmers and agriculturists use drones equipped with infrared sensors that can be tuned to detect crop health, enabling them to react and improve crop conditions locally, with inputs of fertilizers or insecticides.
- **Weather Forecast**: Drones are also being developed to monitor dangerous and unpredictable weather. Since they are relatively cheap and unmanned, drones can be sent into hurricanes and tornadoes so that scientists and weather forecasters acquire new insights into their behavior and trajectory and prevent mishaps.
- **Law Enforcement**: Drones are contributing to the law by helping with the surveillance of large crowds to ensure public safety. They assist in monitoring criminal and illegal activities such as illegal transportation of drugs, human traffic or organized crime.

1.3 Classification of Drones

Depending on the platform and mission, drones often vary widely in their configurations. Therefore, they can be classified into several categories based on different parameters. Adam Watts, classified drones platforms for civil scientific and military uses based upon performance characteristics and features, such as size, flight endurance, wing span and loading, range, maximum altitude and speed, and production cost. In his classification, Watts classified drones as MAVs (Micro of Miniature Aerial Vehicles), NAVs (Nano Aerial Vehicles), VTOL (Vertical Take-Off and Landing), HTOL (Horizontal Take-Off and Landing) LASE (Low Altitude, Short Endurance), LASE Close, LALE (Low Altitude, Long Endurance), MALE (Medium Altitude, Long Endurance) and HALE (High Altitude, Long Endurance). In figure 1.9, the spectrum of different types of drones is presented.

Furthermore, drones can also be categorised into three classes based on the minimum take-off weight combined with how the drones are intended to be used and where they are expected to be operated. The table below show the Brooke-Holland weight categorization:

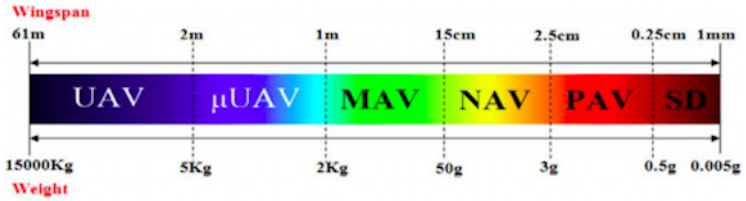


Figure 1.9: Spectrum of drones

Class	Type	Weight Range
Class I(a)	Nano drones	weight $\leq$ 200 g
Class I(b)	Micro drones	200 g $<$ weight $\leq$ 2 kg
Class I(c)	Mini drones	2 kg $<$ weight $\leq$ 20 kg
Class I(d)	Small drones	20 kg $<$ weight $\leq$ 150 kg
Class II	Tactical drones	150 kg $<$ weight $\leq$ 600 kg
Class III	MALE/HALE/Strike drones	weight $\geq$ 600 kg

Table 1.1: Brooke-Holland categorization

- **UAVs**: they are distinguishable from other types of small drones such as MAVs and NAVs by several aspects including the operational purpose of the aerial vehicle, the materials used in its fabrication, and the complexity and cost of the control system. UAVs vary widely in configuration and size. They can also be considered as HTOL, VTOL, hybrid model (tilt-wing, tilt-rotor, tilt-body, and ducted fan).

- **HTOL and VTOL** they are one of UAV's four configurations which are specified by lift/mass balance and by stability and control. There are flying tailless wing UAVs or tailplane-aft, tailplane forward and tail-aft on booms. Different propulsion systems are used for this particular configuration.

- **Tilt-Rotor, Tilt-Wing and Tilt-Body** The VTOL drones have demonstrated their efficiency in hovering flight mode over HTOL, but they have some limitations in cruise speed because of the stalling of the retreating blades. For longer range missions, higher cruise speed is required. However, the ability of vertical take-off and landing is highly valuable. To overcome these limitations, new hybrid model combining the capabilities of both VTOL and HTOL was introduced. Tilt-Rotor UAVs have all rotors initially in vertical position in vertical flight, but for the cruise flight, they tilt forward through 90°. In Tilt-Wing UAVs, the rotors are usually fixed to the wings and tilt with the wings hence the name. The angle of the whole wing is changed from zero to 90° in order to convert its flight modes from horizontal to vertical. The Tilt-Body UAVs or free wing-tilt UAVs are completely different than the two other configurations, the wings are completely free to rotate in pitch axis and the fuselage is a lifting body.

- **Helicopter and heli-wing** There are four types of helicopter UAVs, namely single rotor, coaxial rotor, tandem rotor, and quad-rotor. Heli-wing UAVs are other types of drones which use a rotating wing as their blade. They can fly as a helicopter vertically and also fly as a fixed wing UAV.

- **μUAVs**: Are small unmanned aerial vehicles, small enough to be man-portable. They can be carried and launched by hand. They have exacly the same configurations as the bigger model of UAVs but smaller in size and lighter. The propulsion system generally used in this type of drones is electrical mainly because μUAVs cannot carry big payloads.

Other than HTOL, VTOL and Tilt-rotor, there exist some other variants of μUAVs such as Ornithopter, Flapping wing, Cyclocopter, Ducted fan and other unconventional types.

- **MAV and NAV UAVs**: Micro and Nano unmanned vehicles as the name already suggests, are the smallest and lightest types of UAVs. These drones are mainly used to carry visual, acoustic, chemical and biological sensors, they have a range less than 1 km and a maximum flight altitude around 100 m and medium autonomy. Different type of Micro and Nano air vehicles are attracting various disciplines including Aerospace, mechanical, electrical and computer engineering.
- **Bio-drones** Because of the importance of reconnainssance and patrolling in civil and military applications, applying new instruments for this type of tasks has received much attention. One of the techniques developed for this type of missions, propose the use of live birds and insects that can be controlled by using some electrical chips on them instead of designing new artificial drones.

Figure 1.10: RQ-4 Global Hawk

Figure 1.11: Boeing X-50 Dragonfly

Figure 1.12: UAV Quand Tilt Wing

1.4 Conclusion

In this chapter, we briefly presented the historical context associated with the appearance early design of the first flying aircrafts. We have also seen the major UAV architectures and their applications. Different configurations of vehicles have been mentioned such as: rotary wings, fixed-wings and flapping-wings.

CHAPTER

2

STATE OF THE ART

2.1 Introduction

Several commercial flight control systems for small UAVs are currently available in the market, intended to be customizable for integration with a wide range of unmanned aerial systems They typically include integrated sensors such as GPS receivers, accelerometers, rate gyros, and barometric sensors for measuring the state of the vehicle as well as a processing system to run control loops and communicate with ground solution software. These control system units are typically used for civilian tasks as aerial photography, mapping, research or just as hobby. The next chapter summarizes some of the widely used open-source control systems for drone control.

(a) ArduPilot (b) PX4 AutoPilot (c) OpenPilot Revolution

Figure 2.1: Commercial AutoPilots

2.2 ArduPilot

ArduPilot is an open-source autopilot system that enable the creation and use of trusted, autonomous, unmanned vehicle system and provides a comprehensive suite of tools suitable for any aerial vehicle and application. ArduPilot is a constantly evolving framework thanks to its large community of users. Although ArduPilot does not manufacture any hardware, ArduPilot firmware works on a wide variety of different hardware, and offer some advanced functionalities including real-time communication and Mission Planner which features point-and-click interaction with any hardware supported, custom scripting and complex realistic simulations. Some of other ArduPilot features are presented in the section below:

- Many command modes to fit every type of vehicle: Acro, Stabilize, Loiter, Alt-hold, Return To Launch, Land, Follow Me, GeoFence, etc.
- Autonomous flight modes that execute fully scripted missions with advanced features.
- Advanced failsafe options bring peace of mind in the event of lost control signal, low battery conditions, or other system failures.
- Three Axis camera control and stabilization, shutter control, live video link with programmable on-screen-display.
- Real-time two-way communication between your GCS and controller, including GPS position, battery status, and other live information.
- Full data logging for comprehensive post mission analysis, with graphing and Google Earth mapping tools.
- Industry leading control algorithms for vehicles of all types, with robust sensor compensation algorithms, filtering and tuning capabilities.
- Cross-platform. Supports Windows, Mac and Linux. Use the graphical Mission Planner setup utility in Windows (works under Parallels on a Mac or Mono on Linux) or use a command-line interface on any other operating system. Ground stations are available for all three operating systems. Based on the Arduino programming environment, which is also fully cross-platform.
- Supports full "hardware-in-the-loop" simulation with Xplane and Flight Gear.

2.3 PX4 Autopilot

PX4 is a popular open-source flight control software for drones and other unmanned vehicles. It provides a flexible set of tools for drone developers to share technologies to create tailored solutions for drone applications. PX4 provides a standard to deliver drone hardware support and software stack, allowing an ecosystem to build and maintain hardware and software in a scalable way. PX4 is part of Dronecode, a non-profit organization administered by Linux Foundation to foster the use of open source software on flying vehicles. Dronecode also hosts QGroundControl, MAVLink and the SDK.

- Real-time Kinematic (RTK) is a satellite navigation technique used to enhance the precision of position data derived from satellite-based positioning systems. It increases the accuracy of GNSS/GPS system to centimeter-level.
- Precision Landing and target recognition for multicopters using IR-LOCK sensor. It enables landing with a precision of roughly 10 cm.
- Iridium/RockBlock Satellite Communication System can be used to provide long range high latency link between a ground station and the aerial vehicle.
- Obstacle Avoidance enables the aerial vehicle to navigate around obstacles when following a preplanned path.
- Collision Prevention can be used to automatically slow and stop the aerial vehicle before it can crash into an obstacle.

2.4 OpenPilot Revolution

The OpenPilot Revolution, also called 'Revo' is a new breed of Autopilot using the STM32F4 series, 210MIPS ARM Micro-controller. The board contains a hardware floating point unit (FPU) that allows precise, low-latency processing of real-life measurements using advanced attitude estimation algorithms. The Revolution is a flight control coputer with autopilot, intended for multirotors, helicopters and fixed wings. It's a full 10 DoF with gyroscope, accelerometer, magnetometer and pressure sensors.

- Quick Satellite Searching: it only needs 10 seconds to find up to 6 satellites in open space.
- Build-in Compass with a high refresh-rate up to 10GHz.
- Support for GPS, BD/GLONASS, SBAS.
- Supported antennas Active and Passive.
- Odometer Travelled Distance.
- Noise figure On-Chip LNA (NEO-M8M) and eLNA for Extra Lowest Noise Figure (NEO-M8N/Q).

CHAPTER

3

QUADCOPTER MATHEMATICAL MODEL

3.1 Introduction

The quadcopter is a flying aircraft made up of four rotating motors, holds the electronic board in the middle and the motors at four extremities. Before describing the mathematical model of the quadcopter, it is necessary to introduce the reference coordinates in which we describe the structure and the position. For the quadcopter, it is possible to use two reference systems [3.1]. The first called inertial, is fixed to the earth and the second is mobile and it is attached to the barycenter of the quadcopter. In the scientific literature it is called O_{ABC} system, where ABC stands for: *Aircraft Body Center*.

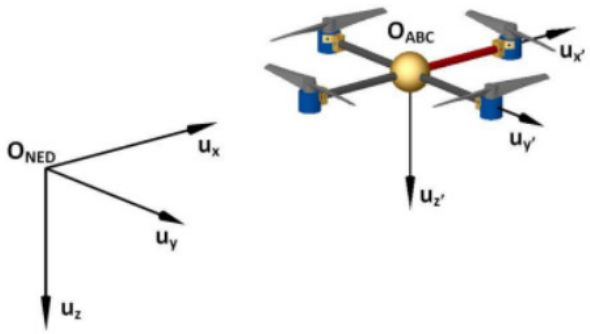


Figure 3.1: Mobile reference system and fixed reference system

3.2 Mathematical Model

The attitude and position of the quadcopter can be controlled to desired values by changing the speeds of the four motors. The set of forces and moments acting on the quadcopter are: the

thrust caused by rotors rotation, the pitching moment and the rolling moment caused by the difference of four robots thrust, the gravity, the gyroscopic effect, and the yawing moment. The gyroscopic effect only appears in the lightweight construction quadcopter. The yawing moment is caused by the unbalance of the four robots rotational speeds, and it can be cancelled out when two rotors rotate in the opposite direction. As a result, the propellers are divided in two groups. In each group there are two diametrically opposite motors that we can easily observe thanks to their direction of rotation. We distinguish the front and rear propellers rotating counterclockwise and the right and left propellers rotating clockwise as seen in the figure below [3.2]

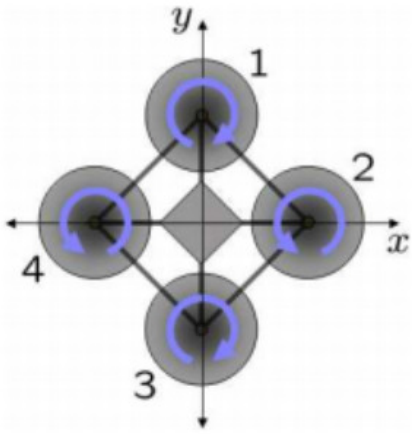


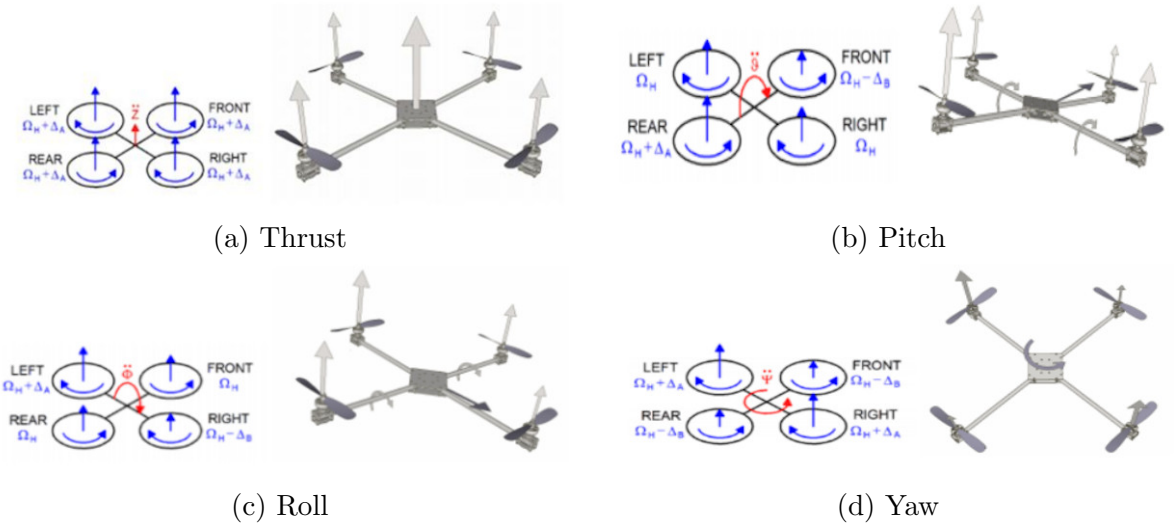
Figure 3.2: Direction of propeller's rotations

The motion of the quadcopter can be divided into two parts, the movement of the barycenter and the movement around the barycenter. Six degrees of freedom are required to describe the full motion, three translation and three rotation motions along three axes. The control for 6 DoF motions can be implemented by adjusting the rotational speeds of different motors. The motions include forward and backward movements, lateral movement, vertical movement, roll motion, pitch and yaw motions. The yaw motion can be realized by a reactive torque produced by the rotors. When the four rotor speeds are the same, the reactive torques will balance each other and the quadcopter will not rotate. Because of four inputs and six outputs, the quadcopter is considered an underactuated nonlinear complex system. In order to control it, some assumptions are made in the process of quadcopter modeling: the quadcopter is a rigid body, the structure is symmetric and the ground effect is ignored.

Depending of the speed rotation of each propeller, it is possible to identify the four basic movements of the quadcopter, which are showed in the figures below [3.3]

3.2.1 Euler Angles

The Euler angles are three angles introduced by Leonard Euler to describe the orientation of a rigid body. To describe such an orientation in a 3-dimensional Euclidean space, three parameters are required. In our case will use ZYX Euler angles. Euler angles are typically denoted as $\phi \in]-\pi, \pi]$, $\theta \in]-\frac{\pi}{2}, \frac{\pi}{2}[$ and $\psi \in]-\pi, \pi]$ and they represent a sequence of three elemental rotations, *i.e* rotations about the axes of a coordinate system, since any orientation can be achieved by composing three elemental rotations. They are also used to describe the orientation of a frame of reference relative to another and they transform the coordinates of a



(a) Thrust (b) Pitch

(c) Roll (d) Yaw

Figure 3.3: Quadcopter movements

point in a reference frame in the coordinates of the same point in another reference frame. The combination used is described by the following rotation matrices:

$$\boldsymbol{R}_x(\phi) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\phi & -\sin\phi \\ 0 & \sin\phi & \cos\phi \end{bmatrix} \tag{3.1}$$

$$\boldsymbol{R}_y(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix} \tag{3.2}$$

$$\boldsymbol{R}_z(\psi) = \begin{bmatrix} \cos\psi & -\sin\psi & 0 \\ \sin\psi & \cos\psi & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{3.3}$$

The inertial position coordinates and the body reference coordinates are related by the rotation matrix $\boldsymbol{R}_{zyx}(\phi, \theta, \psi) \in SO(3)$ [3.5]:

$$\boldsymbol{R}_{zyx}(\phi, \theta, \psi) = \boldsymbol{R}_z(\psi).\boldsymbol{R}_y(\theta).\boldsymbol{R}_x(\phi) \tag{3.4}$$

$$= \begin{bmatrix} \cos\psi\cos\theta & \cos\psi\sin\theta\sin\phi - \sin\psi\cos\phi & \cos\psi\sin\theta\sin\phi + \sin\psi\sin\phi \\ \sin\psi\cos\theta & \sin\psi\sin\theta\sin\phi + \cos\psi\cos\phi & \sin\psi\sin\theta\cos\phi - \cos\psi\sin\phi \\ -\sin\theta & cos\theta\sin\phi & \cos\theta\cos\phi \end{bmatrix} \tag{3.5}$$

This matrix describe the rotation from the body reference system to the inertial reference as shown in the following figure [3.4]:

3.2.2 Newton-Euler Formalism

We provide here a mathematical model of the quadcopter, exploiting Newton and Euler equations for the 3D motion of a rigid body. The goal is to obtain a deeper understanding of the dynamics of the quadcopter and to provide a model that is sufficiently reliable for simulating and controlling its behavior.

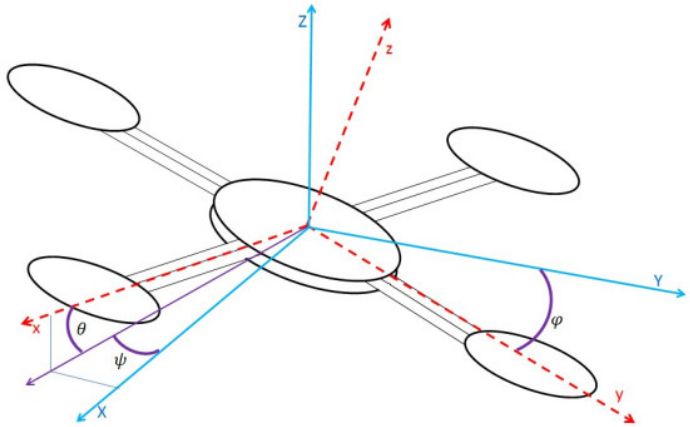


Figure 3.4: Euler Angles

We denote $\chi = \begin{bmatrix} x & y & z & \phi & \theta & \psi \end{bmatrix}^T$ the vector containing the linear and angular position of the quadcopter in the inertial frame and $\mu = \begin{bmatrix} u & v & w & p & q & r \end{bmatrix}^T$ the vector containing the linear and angular velocities in the body frame. From 3D body dynamics, it follows that the two reference frames are linked by the following relations:

$$\boldsymbol{v} = \boldsymbol{R} \,.\, \boldsymbol{v_B} \tag{3.6}$$

$$\boldsymbol{\omega} = \boldsymbol{T} \,.\, \boldsymbol{w_B} \tag{3.7}$$

where $v = \begin{bmatrix} \dot{x} & \dot{y} & \dot{z} \end{bmatrix}^T \in \mathbb{R}^3$, $\omega = \begin{bmatrix} \dot{\phi} & \dot{\theta} & \dot{\psi} \end{bmatrix}^T \in \mathbb{R}^3$, $v_B = \begin{bmatrix} u & v & w \end{bmatrix}^T \in \mathbb{R}^3$, $\omega_B = \begin{bmatrix} p & q & r \end{bmatrix}^T \in \mathbb{R}^3$, and $\boldsymbol{T}$ is a matrix for angular transformations:

$$\boldsymbol{T} = \begin{bmatrix} 1 & \sin\phi\tan\theta & \cos\phi\tan\theta \\ 0 & \cos\phi & -\sin\phi \\ 0 & \sin\phi\cos\theta^{-1} & \cos\phi\cos\theta^{-1} \end{bmatrix} \tag{3.8}$$

The quadcopter kinematic model is:

$$\begin{cases} \dot{x} = u[\cos\psi\cos\theta] + v[\cos\psi\sin\theta\sin\phi - \sin\psi\cos\phi] + w[\cos\psi\sin\theta\sin\phi + \sin\psi\sin\phi] \\ \dot{y} = u[\sin\psi\cos\theta] + v[\sin\psi\sin\theta\sin\phi + \cos\psi\cos\phi] + w[\sin\psi\sin\theta\cos\phi - \cos\psi\sin\phi] \\ \dot{z} = -u[\sin\theta] + v[cos\theta\sin\phi] + w[\cos\theta\cos\phi] \\ \dot{\phi} = p + q[\sin\phi\tan\theta] + r[\cos\phi\tan\theta] \\ \dot{\theta} = q[\cos\phi] - r[\sin\phi] \\ \dot{\psi} = q\,\dfrac{\sin\phi}{\cos\theta} + r\,\dfrac{\cos\phi}{\cos\theta} \end{cases} \tag{3.9}$$

Newton's law of dynamics states the following matrix relation for the total force acting on the quadcopter:

$$m(\boldsymbol{w}_B \wedge \boldsymbol{v}_B + \dot{\boldsymbol{v}}_B) = \boldsymbol{F}_B \tag{3.10}$$

where m is the mass of the quadcopter, $\wedge$ is the cross product and $\boldsymbol{F}_B = \begin{bmatrix} f_x & f_y & f_z \end{bmatrix}^T \in \mathbb{R}^3$ is the total force acting the quadcopter.

Euler's equation on the other hand gives the total torque applied to the quadcopter:

$$\boldsymbol{I}\dot{\boldsymbol{\omega}}_B + \boldsymbol{\omega}_B \wedge (\boldsymbol{I}\boldsymbol{\omega}_B) = \boldsymbol{\tau}_B \tag{3.11}$$

where $\boldsymbol{\tau}_B = \begin{bmatrix}\tau_\phi & \tau_\theta & \tau_\psi\end{bmatrix}^T \in \mathbb{R}^3$ is the total torque acting on the quadcopter and $\boldsymbol{I}$ is the diagonal inertial matrix:

$$\boldsymbol{I} = \begin{bmatrix} I_{xx} & 0 & 0 \\ 0 & I_{yy} & 0 \\ 0 & 0 & I_{zz} \end{bmatrix} \tag{3.12}$$

So, the dynamical model of the quadcopter in the body frame is:

$$\begin{cases} f_x = m(\dot{u} + qw - rv) \\ f_y = m(\dot{v} - pw + ru) \\ f_z = m(\dot{w} + pv - qu) \\ \tau_\phi = \dot{p}I_{xx} - qrI_{yy} + qrI_{zz} \\ \tau_\theta = \dot{q}I_{yy} + prI_{xx} - prI_{zz} \\ \tau_\psi = \dot{r}I_{zz} - pqI_{xx} + pqI_{yy} \end{cases} \tag{3.13}$$

These equations hold as long as we assume that the origin and the axes of the body frame coincide with the barycenter of the quadcopter and the principal axes.

3.3 Forces and moments

The set of total forces acting on a quadcopter are the thrust force produced by the spinning motors which lift it up and the gravitational force due to the earth gravitational field, and lastly the drag force due to the air friction. All the spinning motors create torques about the z-axis, this created torque is mandatory to keep all the propellers spinning and it also provides the necessary thrust to overcome the drag force. The external forces $\boldsymbol{F}_B$ in the body frame are given by:

$$\boldsymbol{F}_B = T_z\hat{\boldsymbol{e}}_3 - mg\boldsymbol{R}^T.\hat{\boldsymbol{e}}_z + \boldsymbol{F}_w \tag{3.14}$$

where $\hat{\boldsymbol{e}}_z$ is the unit vector in the inertial z-axis, $\hat{\boldsymbol{e}}_3$ is the unit vector in the body z-axis, g is the gravitational acceleration, T_z is the total thrust generated by rotors and $\boldsymbol{F}_w = \begin{bmatrix}f_{wx} & f_{wy} & f_{wz}\end{bmatrix}^T \in \mathbb{R}^3$ are the forces produced by wind on the quadcopter. Similarly, the external moments in the body frame $\boldsymbol{\tau}_B$ are given by:

$$\boldsymbol{\tau}_B = \boldsymbol{\tau}_c - \boldsymbol{g}_a + \boldsymbol{\tau}_w \tag{3.15}$$

where $\boldsymbol{g}_a$ represents the gyroscopic moments caused by the combined rotation of the four rotors and the vehicle body, $\boldsymbol{\tau} = \begin{bmatrix}\tau_\phi & \tau_\theta & \tau_\psi\end{bmatrix}^T \in \mathbb{R}^3$ are the control torques generated by differences in the rotor speeds and $\boldsymbol{\tau}_w = \begin{bmatrix}\tau_{w\phi} & \tau_{w\theta} & \tau_{w\psi}\end{bmatrix}^T \in \mathbb{R}^3$ are the torques produced by the drag force. The gyroscopic moment $\boldsymbol{g}_a$ is given by:

$$\boldsymbol{g}_a = \sum_{i=1}^{4} (-1)^{i+1} J_P(\boldsymbol{\omega}_B \wedge \hat{\boldsymbol{e}}_3)\Omega_i \tag{3.16}$$

where J_p is the inertia of each spinning rotor and Ω_i is the angular speed of the i-th rotor. The term J_P is found to be relatively small, as a result, the gyroscopic moments are removed in the controller formulation. In addition, there are numerous aerodynamic and aeroelastic phenomenon that affect the flight of the quadcopter, such as the ground-effect: when flying close to the ground or during the landing stage, the air flow generated by the propellers disturbs the dynamics of the quadcopter.

So the complete dynamic model of the quadcopter using the Newton-Euler formalism in the body frame is obtained substituting the force expression [3.14] in the dynamic model [3.13]:

$$\begin{cases} f_{wx} - mg\sin\theta = m(\dot{u} + qw - rv) \\ f_{wy} + mg[\cos\theta\sin\phi] = m(\dot{v} - pw + ru) \\ f_{wz} + mg[\cos\theta\cos\phi] - T_z = m(\dot{w} + pv - qu) \\ \tau_\phi + \tau_{w\phi} = \dot{p}I_{xx} - qrI_{yy} + qrI_{zz} \\ \tau_\theta + \tau_{w\theta} = \dot{q}I_{yy} + prI_{xx} - prI_{zz} \\ \tau_\psi + \tau_{w\psi} = \dot{r}I_{zz} - pqI_{xx} + pqI_{yy} \end{cases} \tag{3.17}$$

3.4 Actuator Dynamics

Here we consider the inputs that can be applied to the system in order to control the behavior of the quadcopter. We have four rotors, and the DoF we control are as many commonly, the control inputs that are considered are one for the vertical thrust and one for each of the angular motions. Let us consider the values of the input forces and torques proportional to the squared speeds of the rotors, their values are the following:

$$\begin{cases} T_z = b(\Omega_1^2 + \Omega_2^2 + \Omega_3^2 + \Omega_4^2) \\ \tau_\phi = bl(\Omega_3^2 - \Omega_1^2) \\ \tau_\theta = bl(\Omega_4^2 - \Omega_2^2) \\ \tau_\psi = d(\Omega_2^2 + \Omega_4^2 - \Omega_1^2 - \Omega_3^2) \end{cases} \tag{3.18}$$

where l is the distance between any rotor and the center of the drone, b is the thrust factor and d is the drag factor. Substituting [3.18] in [3.17] we obtain the dynamic model of the quadcopter in the body framed:

$$\begin{cases} f_{wx} - mg\sin\theta = m(\dot{u} + qw - rv) \\ f_{wy} + mg[\cos\theta\sin\phi] = m(\dot{v} - pw + ru) \\ f_{wz} + mg[\cos\theta\cos\phi] - T_z = m(\dot{w} + pv - qu) \\ bl(\Omega_3^2 - \Omega_1^2) + \tau_{w\phi} = \dot{p}I_{xx} - qrI_{yy} + qrI_{zz} \\ bl(\Omega_4^2 - \Omega_2^2) + \tau_{w\theta} = \dot{q}I_{yy} + prI_{xx} - prI_{zz} \\ d(\Omega_2^2 + \Omega_4^2 - \Omega_1^2 - \Omega_3^2) + \tau_{w\psi} = \dot{r}I_{zz} - pqI_{xx} + pqI_{yy} \end{cases} \tag{3.19}$$

3.5 State Space

After writing the quadcopter dynamical mode, it is possible to rewrite them in the state-space form, to do so, we begin by organizing the state's vector in the following way:

$$\boldsymbol{x} = \begin{bmatrix} x & y & z & u & v & w & \phi & \theta & \psi & p & q & r \end{bmatrix}^T \in \mathbb{R}^{12} \tag{3.20}$$

and combining the previous equations [3.13] and [3.9] in one form give us:

$$\begin{cases} \dot{x} = u[\cos\psi\cos\theta] + v[\cos\psi\sin\theta\sin\phi - \sin\psi\cos\phi] + w[\cos\psi\sin\theta\sin\phi + \sin\psi\sin\phi] \\ \dot{y} = u[\sin\psi\cos\theta] + v[\sin\psi\sin\theta\sin\phi + \cos\psi\cos\phi] + w[\sin\psi\sin\theta\cos\phi - \cos\psi\sin\phi] \\ \dot{z} = -u[\sin\theta] + v[cos\theta\sin\phi] + w[\cos\theta\cos\phi] \\ \dot{u} = rv - qw - g\sin\theta + \dfrac{f_{wx}}{m} \\ \dot{v} = pw - ru + g[\cos\theta\sin\phi] + \dfrac{f_{wy}}{m} \\ \dot{w} = qu - pv + g[\cos\theta\cos\phi] + \dfrac{f_{wx} - T_z}{m} \\ \dot{\phi} = p + q[\sin\phi\tan\theta] + r[\cos\phi\tan\theta] \\ \dot{\theta} = q[\cos\phi] - r[\sin\phi] \\ \dot{\psi} = q\,\dfrac{\sin\phi}{\cos\theta} + r\,\dfrac{\cos\phi}{\cos\theta} \\ \dot{p} = \dfrac{I_{yy} - I_{zz}}{I_{xx}}rq + \dfrac{\tau_\phi + \tau w\phi}{I_{xx}} \\ \dot{q} = \dfrac{I_{zz} - I_{xx}}{I_{yy}} + \dfrac{\tau_\theta + \tau w}{I_{yy}} \\ \dot{r} = \dfrac{I_{xx} - I_{yy}}{I_{zz}} + \dfrac{\tau_\psi + \tau w\psi}{I_{zz}} \end{cases} \tag{3.21}$$

An alternative formulation to the kinematics model useful for studying the control is rewriting the previous equations [3.9] and [3.13] using Newton's law of dynamics:

$$m\dot{\boldsymbol{v}} = \boldsymbol{R}.\boldsymbol{F}_B \tag{3.22}$$

$$= T_z\boldsymbol{R}.\hat{\boldsymbol{e}}_3 - mg\hat{\boldsymbol{e}}_z \tag{3.23}$$

We will assume that $\begin{bmatrix}\dot{\phi} & \dot{\theta} & \dot{\psi}\end{bmatrix}^T = \begin{bmatrix}p & q & r\end{bmatrix}^T$ holds true for small angles of movement. The dynamics model of the quadcopter in the inertial frame is:

$$\begin{cases} \ddot{x} = \frac{T_z}{m}(\cos\psi\sin\theta\cos\phi + \sin\psi\sin\phi) \\ \ddot{y} = \frac{T_z}{m}(\sin\psi\sin\theta\cos\phi - \cos\psi\sin\phi) \\ \ddot{z} = \frac{T_z}{m}(\cos\theta\cos\phi) - g \\ \dot{p} = \dfrac{\tau_\phi}{I_{xx}} + \dfrac{I_{zz} - I_{yy}}{I_{xx}}\dot{\theta}\dot{\psi} \\ \dot{q} = \dfrac{\tau_\theta}{I_{yy}} + \dfrac{I_{xx} - I_{zz}}{I_{yy}}\dot{\phi}\dot{\psi} \\ \dot{r} = \dfrac{\tau_\psi}{I_{zz}} + \dfrac{I_{yy} - I_{xx}}{I_{zz}}\dot{\phi}\dot{\theta} \end{cases} \tag{3.24}$$

Redefining the state's vector as:

$$\boldsymbol{x} = \begin{bmatrix} x & y & z & \phi & \theta & \psi & \dot{x} & \dot{y} & \dot{z} & p & q & r \end{bmatrix}^T \in \mathbb{R}^{12} \tag{3.25}$$

It is possible to rewrite the equations of motion as affine in control state-space:

$$\dot{\boldsymbol{x}} = \boldsymbol{f}(\boldsymbol{x}) + \sum_{i=1}^{4} \boldsymbol{g_i}(\boldsymbol{x})u_i \tag{3.26}$$

where:

$$\boldsymbol{f}(\boldsymbol{x}) = \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{z} \\ q\,\dfrac{\sin\phi}{\cos\theta} + r\,\dfrac{\cos\phi}{\cos\theta} \\ q[\cos\phi] - r[\sin\phi] \\ p + q[\sin\phi\tan\theta] + r[\cos\phi\tan\theta] \\ 0 \\ 0 \\ g \\ \dfrac{I_{yy} - I_{zz}}{I_{xx}}qr \\ \dfrac{I_{zz} - I_{xx}}{I_{yy}}pr \\ \dfrac{I_{xx} - I_{yy}}{I_{zz}}pq \end{bmatrix} \tag{3.27}$$

and:

$$\boldsymbol{g_1}(\boldsymbol{x}) = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & g_1^7 & g_1^8 & g_1^9 & 0 & 0 & 0 \end{bmatrix}^T \tag{3.28}$$

$$\boldsymbol{g_2}(\boldsymbol{x}) = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{I_{xx}} & 0 & 0 \end{bmatrix}^T \tag{3.29}$$

$$\boldsymbol{g_3}(\boldsymbol{x}) = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{I_{yy}} & 0 \end{bmatrix}^T \tag{3.30}$$

$$\boldsymbol{g_4}(\boldsymbol{x}) = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{I_{zz}} \end{bmatrix}^T \tag{3.31}$$

with:

$$g_1^7 = \frac{1}{m}(\cos\psi\sin\theta\cos\phi + \sin\psi\sin\phi) \tag{3.32}$$

$$g_1^8 = \frac{1}{m}(\sin\psi\sin\theta\cos\phi - \cos\psi\sin\phi) \tag{3.33}$$

$$g_1^9 = \frac{1}{m}(\cos\theta\cos\phi) \tag{3.34}$$

$$\tag{3.35}$$

and the control input vector u is:

$$u = \begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \end{bmatrix} = \begin{bmatrix} T_z \\ \tau_\phi \\ \tau_\theta \\ \tau_\psi \end{bmatrix} = \begin{bmatrix} T_z \\ bl(\omega_3^2 - \omega_1^2) \\ bl(\omega_4^2 - \omega_2^2) \\ d(\omega_2^2 + \omega_4^2 - \omega_1^2 - \omega_3^2) \end{bmatrix} \tag{3.36}$$

In condense form, the quadcopter dynamical model is given as:

$$\begin{bmatrix} \dot{\xi} \\ \ddot{\xi} \\ \dot{\eta} \\ \ddot{\eta} \end{bmatrix} = \begin{bmatrix} v \\ \dfrac{1}{m}T_z\boldsymbol{R}\hat{\boldsymbol{e}}_3 - g\hat{\boldsymbol{e}}_z \\ \dot{\omega} \\ \boldsymbol{I}^{-1}(\boldsymbol{\tau}_B - \omega \wedge \boldsymbol{I}\omega) \end{bmatrix} \tag{3.37}$$

3.6 Classical Control Strategy

The development of non-linear control laws that can achieve the desired performance has been subject of much research. Numerous studies have been conducted on the subject. The representation of aerodynamic forces in precise modeling of the quadcopter dynamics uses terminology (aerodynamic terms, aerodynamic coefficient, air density, apparent surface) that are difficult to understand. In the same way, the knowledge of the physical constants of the vehicle (mass, inertia) is subject to uncertainties. The goal is to synthesize control principles that allow the vehicle to be maintained in the presence of external disturbances.

3.6.1 Design of the Backstepping Controller

The backstepping control is based on a multistep method, and at each stage, a virtual command is generated to ensure that the system converges to its equilibrium state. The stabilization of each synthesis step is ensured by the Lyapunov functions.

A nonlinear control strategy is implemented to stabilize the quadcopter near quasi stationary flight. The altitude of the quadcopter is stabilized by using the vertical force input u_1. The desired roll and pitch angles are generated from the position subsystem to the rotational controller. The rotational controller is used to stabilize the quadcopter near quasi stationary flight with inputs u_2, u_3 and u_4.

Translation Dynmaics

Altitude subsystem of the quadcopter is given by [3.24]. An integral backstepping control is implementing for altitude subsystem. Firstly, we begin by defining the position error δ_ξ and the velocity error δ_v as $\delta_\xi = \xi_d - \xi$ and $\delta_v = \rho_1 - v$ respectively where ρ_1 is a virtual input as the backstepping control strategy requires. So the Lyapunov function candidate is as follows:

$$\mathcal{V}_1 = \frac{1}{2}\delta_\xi^T \delta_\xi \tag{3.38}$$

Differentiating $\mathcal{V}_1$ with respect to time gives:

$$\dot{\mathcal{V}}_1 = \delta_\xi^T \dot{\delta}_\xi \tag{3.39}$$

$$= \delta_\xi^T (\dot{\xi}_d - \dot{\xi}) \tag{3.40}$$

replacing $\dot{\xi}$ with v from the quadcopter model [3.37] and adding and subtracting ρ_1 from the above equation leads to:

$$\dot{\mathcal{V}}_1 = \delta_\xi^T (\dot{\xi}_d - v) \tag{3.41}$$

$$= \delta_\xi^T (\dot{\xi}_d - v) + \delta_\xi^T (\rho_1 - \rho_1) \tag{3.42}$$

$$= \delta_\xi^T (\dot{\xi}_d - \rho_1) + \delta_\xi^T (\rho_1 - v) \tag{3.43}$$

$$= \delta_\xi^T (\dot{\xi}_d - \rho_1) + \delta_\xi^T \delta_v \tag{3.44}$$

By setting $\rho_1 = \dot{\xi}_d - k\delta_\xi$ with $k > 0$ and replacing it above:

$$\dot{\mathcal{V}}_1 = -k\delta_\xi^T \delta_\xi + \delta_\xi^T \delta_v \tag{3.45}$$

It's clear that the first term in negative, so the next step in the control strategy is to show that the second term is also negative. For that, we introduce the second Lyapunov candidate function:

$$\mathcal{V}_2 = \mathcal{V}_1 + \frac{1}{2}\delta_v^T \delta_v \tag{3.46}$$

Taking the time derivative of $\mathcal{V}_2$, we obtain:

$$\dot{\mathcal{V}}_2 = \dot{\mathcal{V}}_1 + \delta_v^T \dot{\delta}_v \tag{3.47}$$

$$= -k\delta_\xi^T \delta_\xi + \delta_\xi^T \delta_v + \delta_v^T \dot{\delta}_v \tag{3.48}$$

We already defined δ_v as the error between the virtual velocity ρ_1 and the actual system's velocity v, so differentiating it with respect to time:

$$\dot{\delta_v} = \dot{\rho}_1 - \dot{v} \tag{3.49}$$

$$= \ddot{\xi}_d - k\dot{\delta_\xi} - \frac{1}{m} T_z \boldsymbol{R}\hat{\boldsymbol{e}}_3 + g\hat{\boldsymbol{e}}_z \tag{3.50}$$

Replacing $\dot{\delta_v}$ in [3.48]:

$$\dot{\mathcal{V}}_2 = -k\delta_\xi^T \delta_\xi + \delta_\xi^T \delta_v + \delta_v^T (\ddot{\xi}_d - k\dot{\delta_\xi} - \frac{1}{m} T_z \boldsymbol{R}\hat{\boldsymbol{e}}_3 + g\hat{\boldsymbol{e}}_z) \tag{3.51}$$

$$= k\delta_\xi^T \delta_\xi + \delta_v^T (\delta_\xi + \ddot{\xi}_d - k\dot{\delta_\xi} - \frac{1}{m} T_z \boldsymbol{R}\hat{\boldsymbol{e}}_3 + g\hat{\boldsymbol{e}}_z) \tag{3.52}$$

Choosing $T_z = m\boldsymbol{R}^{-1}[\delta_\xi + \ddot{\xi}_d - k\dot{\delta_\xi} + g\hat{\boldsymbol{e}}_z + \lambda\delta_v]$ with $\lambda > 0$ and replacing it in above [3.52]:

$$\dot{\mathcal{V}}_2 = -k\delta_\xi^T \delta_\xi - \lambda\delta_v^T \delta_v < 0 \tag{3.53}$$

Which is negative. The result that we obtained is so important because it indicates that the origin is asymptotically stable, which translate to δ_ξ and δ_v tend to 0 as time goes by, it also means that ξ goes to ξ_d and our virtual velocity control ρ_1 goes to the actual system's velocity v

Rotational Dynamics

As we have done with translation dynamics, we will do the same thing for the rotational dynamics. We begin by defining δ_η and δ_ω as $\delta_\eta = \eta_d - \eta$ and $\delta_\omega = \rho_2 - \omega$ respectively. Let's the Lyapunov candidate function $\mathcal{V}_3$ be:

$$\mathcal{V}_3 = \frac{1}{2}\delta_\eta^T \delta_\eta \tag{3.54}$$

Taking the time derivative of $\mathcal{V}_3$ gives:

$$\dot{\mathcal{V}}_3 = \delta_\eta^T \dot{\delta}_\eta \tag{3.55}$$

$$= \delta_\eta^T (\dot{\eta}_d - \dot{\eta}) \tag{3.56}$$

replacing $\dot{\eta}$ with ω from the quadcopter model [3.24] and adding and subtracting ρ_2 from the above equation leads to:

$$\dot{\mathcal{V}}_3 = \delta_\eta^T (\dot{\eta}_d - \omega) \tag{3.57}$$

$$= \delta_\eta^T (\dot{\eta}_d - \omega) + \delta_\eta^T (\rho_2 - \rho_2) \tag{3.58}$$

$$= \delta_\eta^T (\dot{\eta}_d - \rho_2) + \delta_\eta^T (\rho_2 - \omega) \tag{3.59}$$

$$= \delta_\eta^T (\dot{\eta}_d - \rho_2) + \delta_\eta^T \delta_\omega \tag{3.60}$$

By setting $\rho_2 = \dot{\eta}_d - k\delta_\eta$ with $k > 0$ and replacing it above:

$$\dot{\mathcal{V}}_3 = -k\delta_\eta^T\delta_\eta + \delta_\eta^T\delta_\omega \tag{3.61}$$

As we can clearly notice, the first term in $\mathcal{V}_3$ is negative as we desire, but the sign of the second term is unknown. For that we define a second Lyapunov candidate function $\mathcal{V}_4$ as:

$$\mathcal{V}_4 = \mathcal{V}_3 + \frac{1}{2}\delta_\omega^T\delta_\omega \tag{3.62}$$

Taking the time derivative of $\mathcal{V}_4$, we obtain:

$$\dot{\mathcal{V}}_4 = \dot{\mathcal{V}}_3 + \delta_\omega^T\dot{\delta_\omega} \tag{3.63}$$

$$= -k\delta_\eta^T\delta_\eta + \delta_\eta^T\delta_\omega + \delta_\omega^T\dot{\delta_\omega} \tag{3.64}$$

We already defined δ_ω as the error between the virtual velocity ρ_2 and the actual system's velocity ω, so differentiating it with respect to time:

$$\dot{\delta_\omega} = \dot{\rho}_1 - \dot{v} \tag{3.65}$$

$$= \ddot{\eta}_d - k\dot{\delta_\eta} - \boldsymbol{I}^{-1}(\boldsymbol{\tau}_B - \omega \wedge \boldsymbol{I}\omega) \tag{3.66}$$

Replacing $\dot{\delta_\omega}$ in [3.64]:

$$\dot{\mathcal{V}}_4 = -k\delta_\eta^T\delta_\eta + \delta_\eta^T\delta_\omega + \delta_\omega^T(\ddot{\eta}_d - k\dot{\delta_\eta} - \boldsymbol{I}^{-1}(\boldsymbol{\tau}_B - \omega \wedge \boldsymbol{I}\omega)) \tag{3.67}$$

$$= k\delta_\eta^T\delta_\eta + \delta_\omega^T(\delta_\eta + \ddot{\eta}_d - k\dot{\delta_\eta} - \boldsymbol{I}^{-1}(\boldsymbol{\tau}_B - \omega \wedge \boldsymbol{I}\omega)) \tag{3.68}$$

By choosing $\boldsymbol{\tau}_B = \boldsymbol{I}[\delta_\eta + \ddot{\eta}_d - k\dot{\delta_\eta}] + \omega \wedge \boldsymbol{I}\omega + \lambda\delta_\omega$ with $\lambda > 0$ and replacing it in the previous equation, we end up with:

$$\dot{\mathcal{V}}_3.5 = -k\delta_\eta^T\delta_\eta - \lambda\delta_\omega^T\delta_\omega < 0 \tag{3.69}$$

So we have demonstrated the asymptotic stability of the origin *i.e* δ_η and δ_ω which means η tend to approach the desired orientation η_d and our virtual control input ρ_2 approaches the system's actual angular velocity ω.

3.7 Simulation Results

Simulation experiments for the attitude control of the quadcopter are done in order to evaluate the efficiency of hierarchical control using the backstepping control strategy. On the whole dynamic model of the quadcopter, an application of the proposed approach is tested without taking into account external disturbances. The initial position is: $\xi(0) = \begin{bmatrix}0 & 0 & 0.5\end{bmatrix}^T$ and the initial angular orientation is: $\eta(0) = \begin{bmatrix}0 & 0 & 0.3\end{bmatrix}^T$, the linear and angular velocities are set to 0.

The simulation results are presented below:

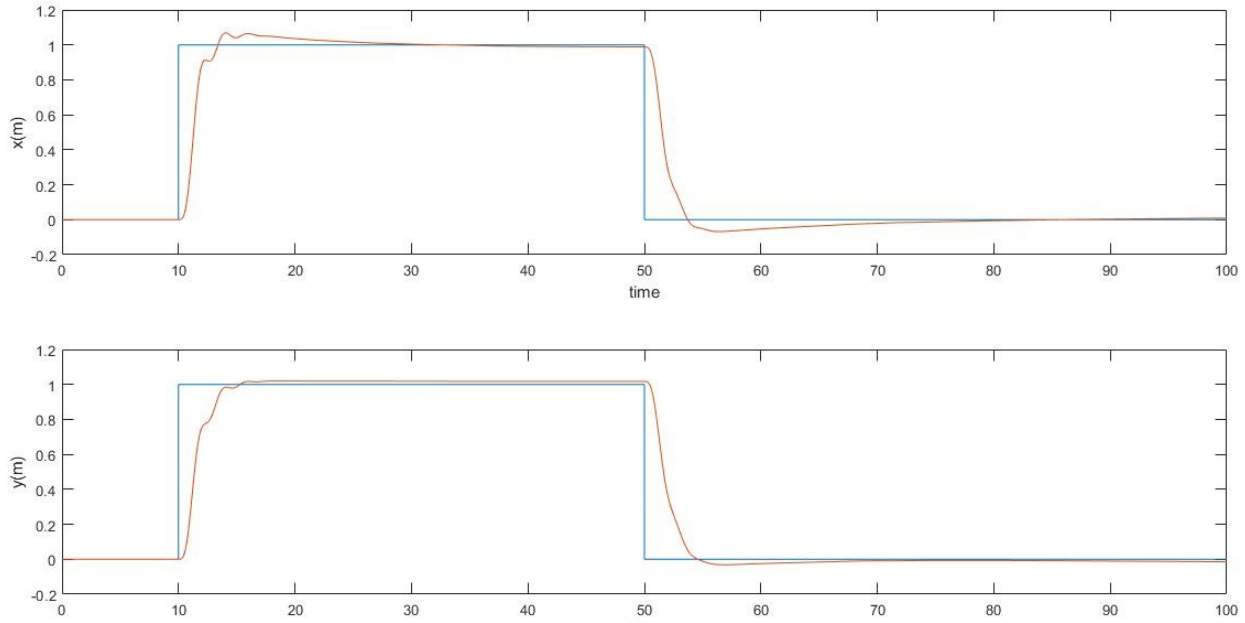


Figure 3.5: x and y positions

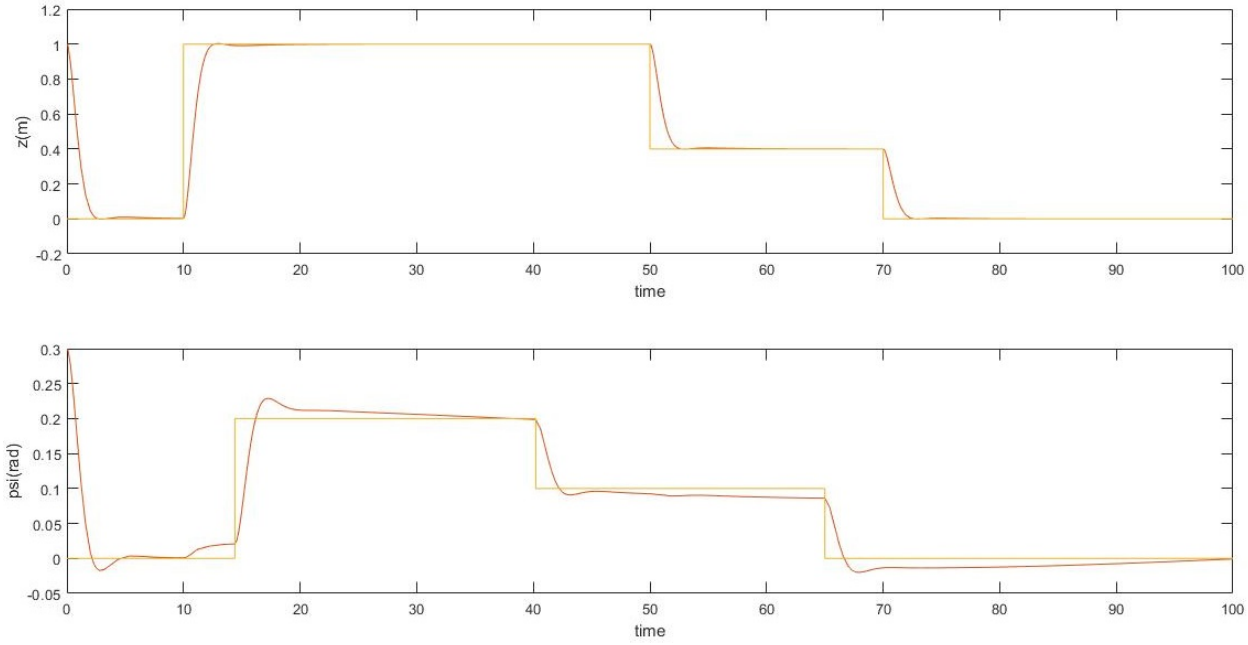


Figure 3.6: z position and yaw angle ψ

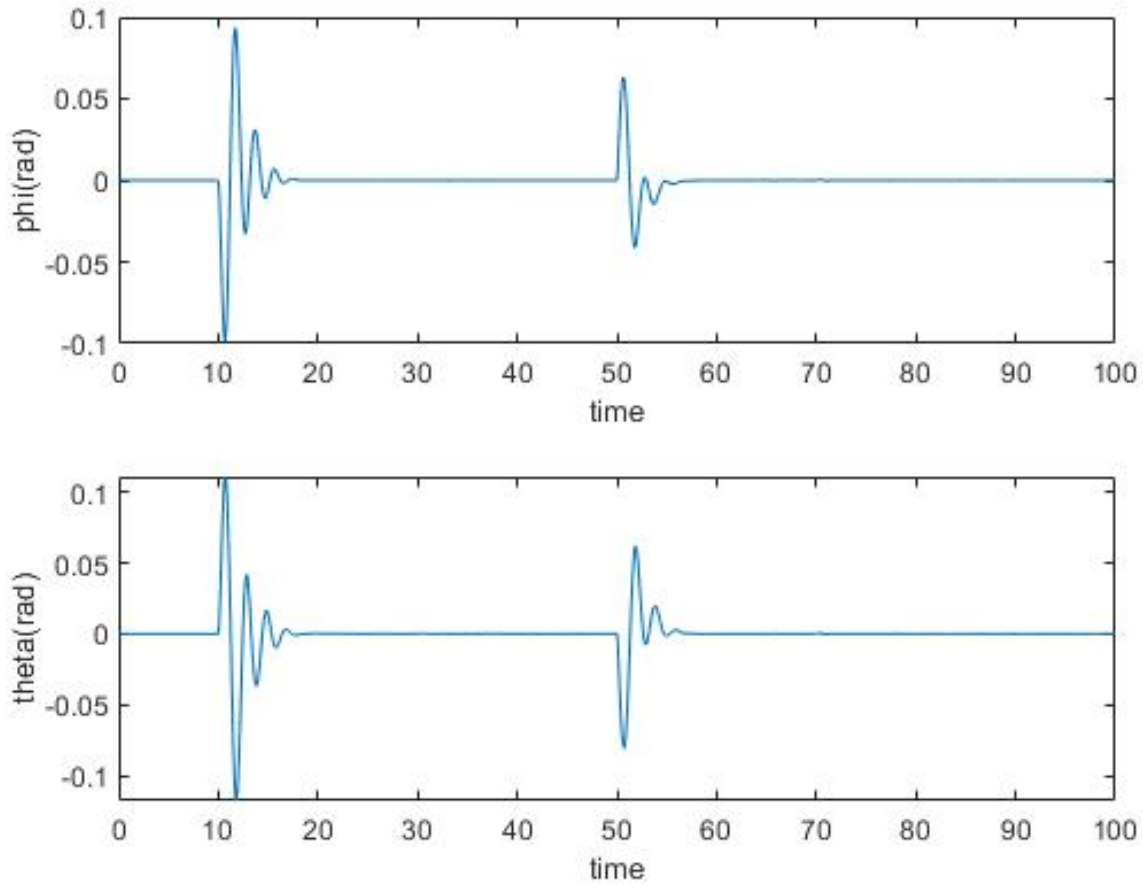


Figure 3.7: Roll and Pitch angles ϕ, θ

The figures above show the evolution of the translational and yaw position over time. When the desired references are introduced, the quadcopter is able to track the reference trajectories. The evolution of quadcopter x, y, and z positions are represented on figures 3.5, 3.6. We see that the trajectory tracking capability is acceptable and the quadcopter maintained the desired attitude. The roll and pitch trajectories are presented in the figure 3.7. These two trajectories were generated by the translational controller to help the flying vehicle following the position trajectory ξ_d. This shows the effectiveness of the proposed control strategy. We note that, when the quadcopter is changing position, we notice some spikes in the roll and pitch angles and this primarily due to the coupling between the quadcopter translation and rotation dynamics as seen in the model dynamics [3.24]

3.8 Conclusion

In this chapter we studied mathematical modelling and control of the quadcopter. The mathematical model was presented and the differential equations were derived from the Newton-Euler formalism. The model was then verified by simulating the flight of the quadcopter with Matlab/Simulink. Stabilisation of attitude of the quadcopter was done by utilising the backstepping control strategy. The simulation proved the presented mathematical model to be realistic in modeling the position and attitude of the quadcopter. The simulation results also showed that the controller was efficient is stabilising the quadcopter to the desired altitude and attitude.

The presented model and control methods were tested only with simulations. Real experimental prototype of a quadcopter should be constructed to achieve more realistic and reliable results.

CHAPTER

4

REINFORCEMENT LEARNING

4.1 Introduction

Reinforcement learning is a highly promising area of Machine Learning paradigms alongside with Supervised learning and unsupervised learning, it deals with how intelligent agents learn and behave in known or unknown, fully observable or partially observable environment in order to maximize the notion of a cumulative reward, it is used primarily in decision-making. The agent is not necessarily a software entity, it could also be embodied in hardware such as robots, UAVs or autonomous vehicles

Reinforcement learning differs from supervised learning in not needing labeled data for training or testing, instead it focuses in finding a balance or trade-off between exploration (exploring uncharted areas in the environment) and exploitation (the current knowledge).

The environment has a state that can be partial of fully observable. The agent can then perform a set of actions to interact with its environment, by taking either a fully deterministic or stochastic action, the environment will then transition to a new state generating a scalar reward. The goal here for the agent is to maximize the cumulative reward and find the optimal or nearly-optimal policy that will help the agent decide which action to take given a state. [2]

The environment is typically stated in the form of **Markov Decision Process** or MDP because many reinforcement learning algorithms for this context use **Dynamic Programming** and they obey the so called: Markov property which states that: The future is independent of the past given the present. Once the current state in known, the history of information encountered so far may be thrown away, and that state is a sufficient statistic that gives us the same characterization of the future as if we had the whole history.

One of the main issues in Reinforcement Learning is finding the right balance between Exploration and Exploitation. Since exploration is inherently costly in terms of resource, time and opportunity, a natural and crucial question in Reinforcement Learning is to address the

dichotomy between exploration of uncharted territory and exploitation of existing knowledge. Such question exists in both the stateless Reinforcement Learning settings such as multi-armed bandit problem and the more general multi-state Reinforcement Learning settings. Specifically, the agent must balance between greedily exploiting what has been learned so far to choose actions that yield near-term higher rewards, and continuously exploring the environment to acquire more information to potentially achieve long-term benefits. Extensive studies have been conducted to find strategies for the bets trade-off between exploration and exploitation. [[18]]

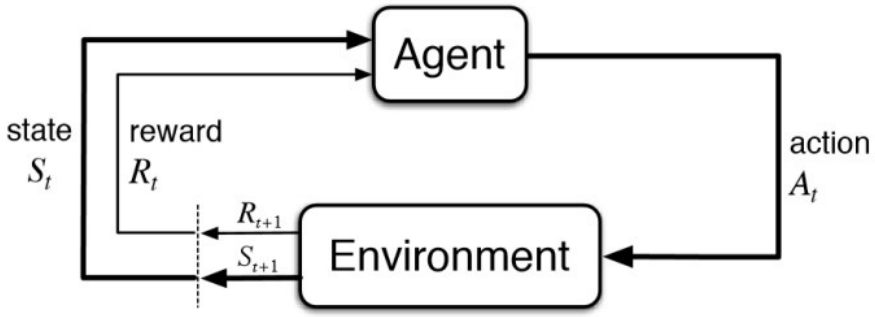


Figure 4.1: Reinforcement learning block diagram

4.2 Reinforcement Learning from Control Theory perspective

Optimal control deals with the problem of finding an optimal control law for a given system to achieve some optimality criterion. A control problem includes the notion of a cost functional that is a function of control and state variables. By using the **Pontryagin's maximum principle** (PMP) or solving the **Hamilton-Jacobi-Bellman** we can derive the optimal control that minimizes or maximizes the cost functional by solving out a set of differential equations describing the path of the control variables. Control problems usually include ancillary constraints.

We can illustrate this with a simple example, consider the problem of cruise control, the question is how should the driver press the acceleration pedal in order to minimize the fuel consumption, given that it must complete a given course in a time not exceeding some amount.

Let's consider the nonlinear discrete-time system illustrated in the figure below with x_k is the state of the system at the k-th sampling time, u_k is the control input to drive the dynamics of the system and w_k is a random disturbance. In generalized and more abstract forms, we can define the cost functional as:

$$\mathcal{J} = g_N(x_N) + \sum_{k=0}^{N-1} g_k(x_k, u_k, w_k) \tag{4.1}$$

subject to the system's dynamics: $x_{k+1} = f_k(x_k, u_k, w_k)$ given the initial condition $x_{k=0} = x_0$, g_N and g_k are referred to as Mayer term and Lagrangian respectively in calculus of variation.

With a full and perfect knowledge of the system state, we can then find the optimal control

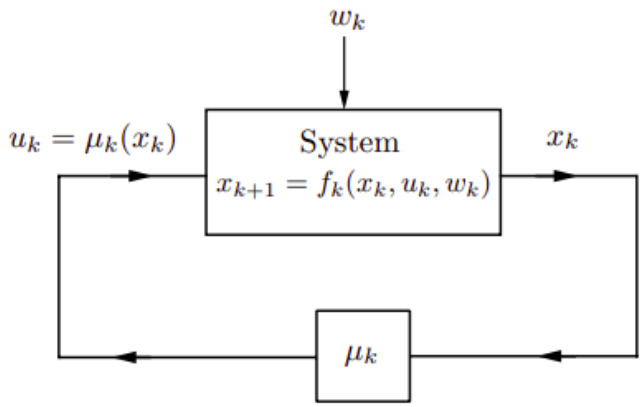


Figure 4.2: discrete-time nonlinear system

law u^* that minimizes the above cost functional, in other words:

$$\min_{u_k} \mathcal{J} = \min_{u_k} \mathbb{E}_{w_k}[g_N(x_N) + \sum_{k=0}^{N-1} g_k(x_k, u_k, w_k)] \tag{4.2}$$

where $\mathbb{E}[.]$ denotes the expected value.

By the help of optimization theory, the optimal control law u^* for our optimal control problem is given by:

$$u_k^* = \arg\min_{u_k} \mathbb{E}_{w_k}[g_N(x_N) + \sum_{k=0}^{N-1} g_k(x_k, u_k, w_k)] \tag{4.3}$$

In Reinforcement Learning context however, we can think of the agent as the controller driving the system, the environment as the plant or the system's dynamics, the reward function as the feedback loop and the cost functional as the value function. Formally, Reinforcement Learning problems can be described as Markov-decision Process (MPD), for simplicity we will assume a fully deterministic environment where a certain action in a given state will consistently result in a next state and reward, so at every time step t:

- The environment is in state s in the state space $\mathcal{S}$ which may be discrete or continuous, starting from the initial state s_0 to the terminal state s_T
- The agent will take an action a in the set of action space $\mathcal{A}$ by obeying the policy $\pi(a_t|s_t)$, $\mathcal{A}$ here can also be discrete or continuous.
- The environment then will transition to a new state s_{t+1} using the state transition dynamics $\mathcal{T}(s_{t+1}|s_t, a_t)$, the next state is only dependent on the current state and action and $\mathcal{T}$ is not known to the agent.
- The agent then receives a reward from the environment using the reward function $r_{t+1} = \mathcal{R}(s_t, a_t, s_{t+1})$ with $r : \mathcal{A} \times \mathcal{S} \to \mathbb{R}$ is a map from action space and state space to a real number resulting in either a reward (positive reinforcement) if $r_t > 0$ or punishment (negative reinforcement) if $r_t < 0$. The reward function $\mathcal{R}$ is not known to the agent.
- The future rewards are discounted by a factor of γ^k where $\gamma \in [0, 1)$ is the discount factor and k is the future timestep.

- Horizon H, is the number of timesteps T needed to complete one episode from s_0 to s_T.

As we said earlier, the main goal of the agent is to find the optimal policy $\pi^*(a_t|s_t)$ that maximizes the discounted cumulative reward $\mathcal{R}(s_t, a_t, s_{t+1})$ in other words:

$$\pi^* = \arg\max_{\pi} \mathbb{E}[\mathcal{R}(s_t, a_t, s_{t+1})|\pi] \tag{4.4}$$

Which is very similar to the optimal control law u^* in classical optimal control.

4.3 Dynamic Programming

The term Dynamic Programming (DP) refers to a collection of algorithms that can be used to compute optimal policies given a perfect model of the environment as a Markov Decision Process MDP. Classical Dynamic Programming algorithms are of limited utility in Reinforcement Learning both because of their assumption of a perfect model and because of their great computational expense, but they are still important theoretically. [16]

The key idea of Dynamic Programming, and of Reinforcement Learning generally, is the use of value functions to organize and structure the search for good policies. In the next section, we show how Dynamic Programming can be used to compute value function and then obtain the optimal policy the agent should take to maximize the return. [16]

4.3.1 Value Iteration

Q-Learning

Q-Learning [19] is a form of model-free Reinforcement Learning, it can be viewed as a method of Asynchronous Dynamic Programming (ADP). It provides agents with the capability of learning to act optimally in Markovian Decision processes by experiencing the consequences of actions, without requiring them to build maps of the domains.

An important question that one may ask is that, if the Reinforcement Learning problem is to find the optimal policy π^* that maximizes the discounted cumulative reward, how does the agent learn by interacting with the environment? The equation for finding the optimal policy does not explicitly indicate which action to try by the agent and the succeeding state to compute the return. So it is interesting to define a function that shows us how good a certain action is, given a particular state, for an agent following a policy. In other words, instead of finding the policy that maximizes the value for all states in any given finite MDP, the equation [4.5] looks for the action that maximizes the quality Q-value for all states and hence the optimal policy π^*.

$$\pi^* = \arg\max_{a} Q(s, a) \tag{4.5}$$

One of the key properties of the Q-value function is that it must satisfy Bellman Optimality Equation, according to which the optimal Q-value for a given state-action pair equals the maximum reward the agent can get from an action in the current state, plus the maximum

discounted reward it can obtain from any possible state-action pair that follows. The Bellman equation is given below:

$$Q^*(s,a) = \mathbb{E}[\mathcal{R}_{t+1} + \gamma \max_{a'} Q^*(s',a')] \tag{4.6}$$

The Bellman equation is the core of the Q-value function algorithm, and the foundation not only in Reinforcement Learning but also in much more general Dynamic Programming, which is a widely used method for solving practical optimization problems. As you may notice, this definition is recursive, intuitively, it decomposes the value computation into an immediate expected reward from the next state $\mathcal{R}_{t+1}$, plus the value of a successor state $Q^*(s',a')$ with a discount factor γ.

Without knowing anything about the dynamics of the environment, the agent tries an action a, observes what happens in the form of reward r and next state s, $\max_{a'} Q^*(s',a')$ chooses the next best action the agent should take that will give the maximum Q-value for the next state, then updating the Q-value for that current state-action pair. Doing the update iteratively will eventually learn the Q-value function.

After N steps into the future, the agent will decide what action to take that yields a better reward resulting in a new state. The weights for this step is calculated as γ^N which value the earlier rewards received by the agent higher than those received later, reflecting the notion of a "good start". The algorithm therefore, has to calculate the quality $Q(s,a)$ of a state-action pair combination. Before learning begins, Q is initialized to a possibly arbitrary fixed value, and the actions are taking randomly to force the agent to explore the environment. As the learning progresses, at each time t, the agent selects an action a_t, enters a new state s_{t+1}, and observes the reward obtained $\mathcal{R}_t$ and the Q-value function get updated using the Bellman equation. In an algorithmic fashion, the Q-Learning process is illustrated in the algorithm below:

Algorithm 1 Q-Learning Algorithm

Initialize $Q(s,a)$ arbitrarily
for for each episode **do**
  Initialize $\mathcal{S}$
  for for each step of episode **do**
    Choose a from s using policy derived from Q
    Take action a, observe $\mathcal{R}$ and s'
    Update
    $Q(s,a) \leftarrow Q(s,a) + \alpha[\mathcal{R} + \gamma \max_{a'} Q(s',a') - Q(s,a)]$
    $s \leftarrow s'$
  until s is terminal
end for

where, α is the learning step or step size that determines to what extent newly acquired information overrides old information. When setting $\alpha = 0$, the agent will learn nothing, or exclusively exploiting prior knowledge and not acquiring new experiences, while for $\alpha = 1$, the agent will only consider the most recent information acquired neglecting past experiences to explore possibilities.

Deep Q-Learning

The iterative nature of the Q-Learning algorithm might be sufficient for tasks where there is relatively small state space, but the method's performance will decrease drastically when we

tackle more complex environments. The big amount of computational resources and time it needs to traverse the new states and modify the Q-values will most likely render the task computationally inefficient and infeasible. An alternative way consists of using a general function approximator to estimate the optimal Q-value function instead of computing the Q-values directly through value iteration, and the method of choice to do it is by using artificial Neural Networks. The act of integrating artificial Neural Network into the Q-learning process is referred to as Deep Q-Learning, and the network that uses Deep Neural Networks with many neurons and hidden layers to approximate Q-functions is called Deep Q-Network or DQN.

The working principle of Deep Q-Learning algorithm is that, a Neural Network receives states from an environment as an input and then produces estimated Q-values for each action the agent can choose in those states. As mentioned earlier, the Q^* must satisfy the Bellman optimality criterion $Q^*(s,a) = \mathbb{E}[\mathcal{R}_{t+1} + \gamma \max_{a'} Q^*(s',a')]$, so we will compute the exact target values from the right-hand side of the equation, then we will compare it to the model's estimated output to calculate the loss. Next, using the Backpropagation algorithm and Stochastic Gradient Descent (SGD), the Neural Network will update its weights and biases by backpropagating the error, and adjust the output once again to minimize the error as much as possible.

It should be noted that, for small state spaces and less complex environments, it is recommended to use standard Q-Learning rather than Neural Networks as conventional value iteration is likely to converge to optimal values faster.

The figure [4.3] below, shows the difference between classical Q-Learning algorithm for Markovian Decision processes, and the Deep Q-Network algorithm.

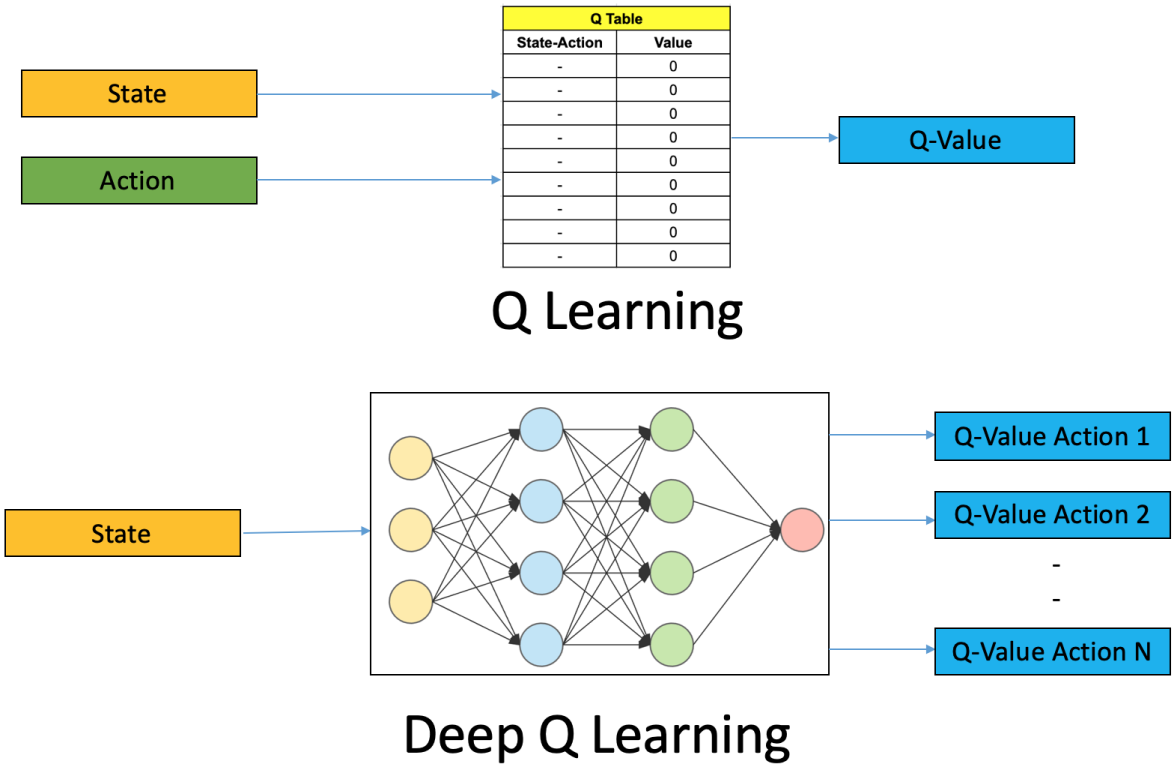


Figure 4.3: Q-Learning vs Deep Q-Network

Double Q-Learning

In some finite stochastic Markovian Decision processes, the well-known Reinforcement Learning algorithm Q-Learning performs very poorly. This poor performance is caused by large overestimations of action values. These overestimations result from a positive bias that is introduced because Q-Learning uses the maximum action value as an approximation for the maximum expected action value. An alternative way is to approximate the maximum expected value for any set of random variables [7]. This is done by using two identical Neural Network models. One learns during experience replay, just like DQN does, and the other one is a copy of the last episode of the first model. The Q-value is actually computed with the second model with the reward added to the next stage maximum Q-value. If every time the Q-value calculates a high number for a certain state, the value that is obtained from the output of the Neural Network for that specific state will become higher. The output value estimated by the output neuron will eventually get higher. Now let suppose that for a state s, the action a is a higher value than action b, then the action a will get chosen every time for that same state s. Now consider if for some memory experience action b becomes the better action for the state s, then it is difficult to tell the Neural Network that the action b is the better action in some condition, since the Network was trained in a way to give a much higher value for action a when given the state s. This justifies the use of a secondary model that is the exact copy of the main model from the last episode and obviously, since the difference between values of the second model are lower than the main model, we use this second model to attain the Q-value. The obtained double estimator sometimes underestimate rather than overestimate the maximum expected value. When finding the index of the highest Q-value from the main model, it will further be used to obtain the action from the second model, and the rest is history. Applying this double estimator to the conventional Q-learning process, constructs a new off-Policy Reinforcement Learning algorithm called Double Q-Learning. [13]

4.3.2 Policy Iteration

We explained in the previous chapter, what is Q-Learning and what it is used for, and we tackled some of the most modified Q-Learning algorithms that use Deep Neural Networks to approximate the Q-value function such as Deep Q-Network and Double Q-Network. The central idea in Q-Learning process is the value of the state-action pair, it is defined as the discounted total reward that the agent can gather from a particular state by doing a particular action. The agent should then act greedily in terms of the income reward guaranteed at the end of every episode. The Value Iteration process just obeys the Bellman optimality criterion. After updating the Q-value function, the policy dictating to the agent how to behave in every state can be obtained with ease by using the equation $\pi(s) = \arg\max_a Q(s, a)$.

There are several reasons why the policy is an interesting topic to explore, at the end of the day, Reinforcement Learning is all about decision-making, So the policy that guarantees the best actions to choose that maximises the reward is what we are looking for. Another reason to go with policies rather than Q-value function is due to complex environments with lots of discrete or continuous action spaces. To be able to decide on what action to take when having $Q(s, a)$, we need to solve the optimization problem that finds the optimal action a that maximizes the quality of being at a particular state, or $Q(s, a)$. In Markovian Decision processes with relatively small number of discrete actions, this is not a problem, we just need to approximate the value of all actions and take the action with the largest Q, but when dealing

with more complex environments with continuous actions such as the speed of a rotating motor, or the angle at which the cruise pedal should be to have an optimal fuel consumption, this optimization problem becomes hard as Q is usually represented by a highly non-linear Neural Network, so finding the argument that maximizes the function's values can be infeasible. In such cases, it is better to avoid values and work with the policy directly. Another convenience of policy learning is when we have stochastic Markovian Decision processes. [12]

In Q-Learning, the Q-values associated to each action were parametrized by a Neural Network that takes the agent state, then the agent will choose the action with the highest Q-value [4.3]. If we want our Neural Network to parametrize the actions, we can either return the index or identifier of the action in Markovian Decision Processes with discrete action space, or return the probability distribution of our actions. For example, for N mutually exclusive actions, the return is a number representing the probability of taking a particular action given a state which the Neural Network will have as an input. This is a much common solution used in finite, stochastic, partially observed Markovian Decision Processes.

Such representation of actions as probabilities has the benefit of smooth representation, meaning if we slightly change the weights and biases of the Neural Network, the output will marginally get changed. In contrast, in discrete action spaces, a small tweak in the Neural Network parameters, can lead to a jump to a different action. This intrinsic property in policy iteration makes it suitable for Parameter-based Value Functions (PVFs) whose inputs include the policy parameters, and the outputs get improved when adjusting the Neural Network parameters. On of the most popular approaches is Policy Gradient. The figure [4.4] illustrates the Policy Iteration process.

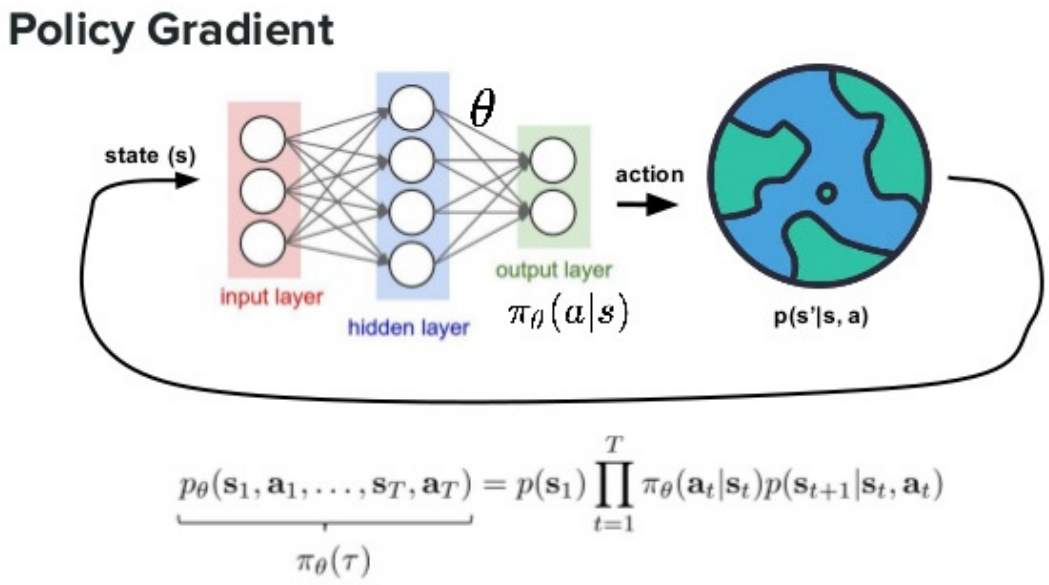


Figure 4.4: Policy Iteration architecture

Policy Gradient

As we have seen in the first chapter, the objective of a Reinforcement Learning agent is to maximize the expected reward when following a policy $\pi(s)$. Like any Machine Learning setup,

we define a set of parameters θ, and proceed by sampling a stochastic policy and adjusting the policy parameters in the direction of greater cumulative reward.

We define the policy gradient as $\nabla J = \mathbb{E}[Q(s,a)\nabla \log \pi(a|s)]$. The scale of the gradient is proportional to the value of the action taken, which is $Q(s,a)$, and the gradient itself is equal to the gradient of the log probability of the action taken. This means that we tend to increase the probability of the actions that have given us good total reward and decrease the probability of actions with bad outcomes. [12] From a practical point of view, Policy Gradient method could be implemented by performing optimization of the loss function defined as: $\mathcal{L} = -Q(s,a)\log \pi(a|s)$. The minus sign indicates that during Stochastic Gradient Descent (SGD), the loss function tends to minimize, however in policy gradient, we want our policy to maximize.

4.3.3 Actor-Critic

As we saw previously with Value and Policy Iteration, the vast majority of Reinforcement Learning programming methods try to overcome difficulties with learning procedures, such as the dimension of the state space, the nature of the action (discrete or continuous), stochasticity and randomness in Markovian Decision Processes, by combining simulation-based learning and compact representations of policies and value function. They fall into one of the following two categories:

- Actor-only method work with a parameterized family of policies. The gradient of the performance, with respect to the actor parameters is directly estimated by simulation, and the parameters are updated in a direction of improvement. A possible drawback of such methods is that the gradient estimators may have a large variance. Furthermore, as the policy changes, a new gradient is estimated independently of past estimates. Hence, there is no "learning" in the sense of accumulation and consolidation of older experiences and informations.
- Critic-only methods rely exclusively on value function approximation and aim at learning an approximate solution to the Bellman equation, which will then hopefully prescribe a near-optimal policy. Such methods are indirect in the sense that they do not try to optimize directly over a policy space. A method of this type may succeed in constructing a good approximation of the value function yet lack reliable guarantees in terms of near-optimality of the resulting policy.

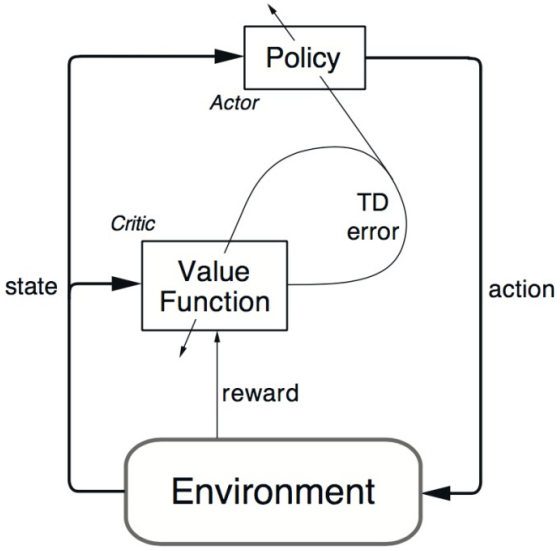


Figure 4.5: Actor-Critic Block Diagram

So Actor-Critic methods aim at combining the strong points of actor-only and critic-only methods. The critic uses an approximation architecture and simulation to learn a value function, which is then used to update the actor's policy parameters in a direction of performance improvement. Such methods, as long as they are gradient-based, may have desirable convergence properties, in contrast to critic-only methods for which convergence is guaranteed in rather limited settings. They also hold the promise of delivering faster convergence due to variance reduction than actor-only methods. On the other hand, theoretical understanding of actor-critic methods has been limited to the case of lookup table representations of policies and value functions.[11]

Below, is the pseudocode for Q-Actor-Critic:

Algorithm 2 Q-Actor-Critic

Initialize parameters s, θ, w and learning rates α_θ, α_w
sample $a \sim \pi_\theta(a|s)$
for t=1...N **do**
  Sample reward $r_t \sim \mathcal{R}(s,a)$ and next state $s' \sim P(s'|s,a)$
  Then sample the next action $a' \sim \pi_\theta(a'|s')$
  Update the policy parameters:
    $\theta \leftarrow \theta + \alpha_\theta Q_w(s,a)\nabla_\theta \log \pi_\theta(a|s)$
  Compute the correction Temporal Difference error for the action-value at time
    $\delta_t = r_t + \gamma Q_w(s',a') - Q_w(s,a)$
  Use the Temporal Difference to update the parameters of Q function
    $w \leftarrow w + \alpha_w \delta_t \nabla_w Q_w(s,a)$
  Move to $a \leftarrow a'$ and $s \leftarrow s'$
end for

4.4 Conclusion

In this chapter, we introduced the concept of Reinforcement Learning as a computational approach to understanding and automating goal-direct learning and decision making. It uses

the framework of Markov Decision Processes as a way to enable interaction between a learning agent and its environment to achieve long-term goals. We also introduced the concept of value function as the agent's capability of learning to act optimally by experiencing the consequences of the actions he takes. In the same context of Q-Learning, we explained the working principle of the most popular algorithms, Deep Q-Network (DQN), and Double Q-Network (DDQN), which utilize Artificial Neural Networks as function approximators to predict optimal Q-values and use them to determine action selections. Later in the chapter, we considered methods that learn a parameterized policy that enable actions to be taken without consulting action-value estimates as an alternative to Q-Learning. They learn and update the policy parameters on each step in the direction of an estimate of the gradient of performance with respect to policy parameters. Finally, we emphasized the significant potential of Policy-Gradient method over action-value methods, especially in complex stochastic environment, and continuous set of states and actions.

CHAPTER

5

QUADCOPTER CONTROL USING REINFORCEMENT LEARNING

5.1 Introduction

Over the past decade, there has been an uptrend in the popularity of Unmanned Aerial Vehicles UAVs. In particular, Quadcopters have received significant attention in the research community, where a significant number of seminal results and applications has been proposed and experimented. This recent growth is primarily attributed to the drop in cost of on-board sensors, actuators, and small-scale embedded computing platforms. Despite the significant progress, flight control is still considered an open research topic. On the one hand, flight control inherently implies the ability to perform highly time-sensitive sensory data acquisition, processing, and computation of forces to apply to the flying aircraft actuators. On the other hand, it is desirable that UAV flight controllers are able to tolerate faults, adapt to changes in the payload and/or the environment, and optimize flight trajectory [10].

The development of intelligent flight control systems is an active area of research, specifically through the use of Artificial Neural Network (NN), which are an attractive option given that they are universal function approximators and resistant to noise [10].

Online learning methods have the advantage of learning the aircraft dynamics in real-time. The main limitation with online learning is that the flight control system is only knowledgeable of its past experiences. It follows that its performances are limited when exposed to a new event. Training models offline using Supervised Learning is problematic, as data is expensive to obtain and derived from inaccurate representations of the underlying aircraft dynamics, which can lead to suboptimal control policies. To construct high-performance intelligent flight control systems, it is necessary to use a hybrid approach. First accurate offline models are used to construct a baseline controller, whereas online learning provides fine tuning and real-time adaptation.

An alternative to supervised Learning for creating offline models is known as Reinforcement Learning (RL). In Reinforcement Learning, as we have seen in the last former, an agent is given a reward for every action it takes in an environment, with the objective to maximize the rewards over time. Using Reinforcement Learning, it is possible to develop optimal control policies for UAVs without making any assumptions about the aircraft dynamics like Value and Policy Iterations (see Chapter 4) Recent work has shown Reinforcement Learning to be effective for UAV autopilot, mentioned in [2] providing adequate path tracking like ArduPilot, OpenPilot Revolution and PX-4. [10]

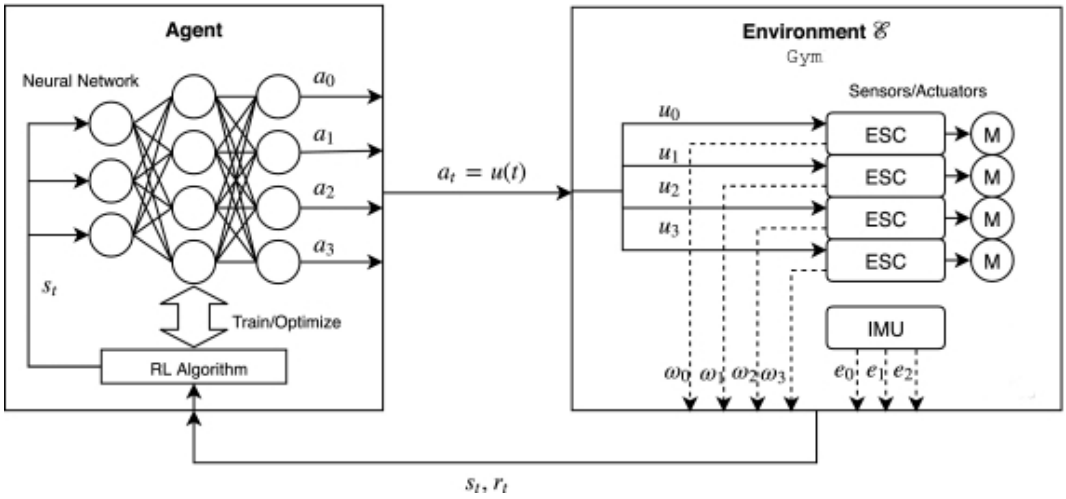


Figure 5.1: Quadcopter control using RL

5.2 Open challenges in Reinforcement Learning for attitude control

Reinforcement Learning is currently being applied to a wide range of applications, each with its own set of challenges. Below, we emphasize several challenges encountered in quadcopter control using Reinforcement Learning:

- **Precision and Accuracy** Many Reinforcement Learning tasks can be solved in a variety of ways. In the case of optimal attitude control, there is small tolerance and flexibility as to the sequence of control signals that will guarantee the desired attitude of the flying drone. Even the slightest deviations can lead to instabilities. Therefore, achieving good level of accuracy and precision is crucial in establishing if Reinforcement Learning is suitable for attitude flight control.

- **Robustness and Adaptation** From control system perspective, robustness refers to the impeccable performance of the controller in the presence of uncertainty and external noise ad disturbances, whereas adaptiveness refers to the controller's ability to adapt to the uncertainties by adjusting its parameters. As it probably known, even in the most accurate simulation environment, the Reinforcement Learning agent will still struggle with uncertainties when the model get transferred to physical hardware. However, it remains unknown in what range of uncertainty the controller can safely operate before adaptation is necessary.

- **Reward Engineering** In Reinforcement Learning, reward shaping is the process of designing a reward system for the agent showing him the return of his actions, whether he did the right thing or not. In the context of attitude control, the reward must encapsulate the agent's performance in achieving the desired attitude goals. As goals become more complex and demanding, identifying which performance metrics are most expressive will be necessary to push the performance of intelligent control system trained with Reinforcement Learning. [10]

In this work, we provide both machine learning and robotics communities with a simple, compact, open-source OpenAI GYM-style environment for simulating and controlling quadcopters, and also training single or multi-agent, model-free or model-based Reinforcement Learning. The code utilized in this project is heavily inspired from the `gym-pybullet-drones` Github repository.

5.3 Related Work

Notable work by Diereks and Jagannathan [4] proposes an intelligent flight control system constructed with Neural Networks to learn the Quadcopter dynamics, online, to navigate along a specified path. This method allows the aircraft to adapt in real-time to external disturbances and unmodeled dynamics. We mention also the outstanding contribution of the Reinforcement Learning pioneers Andrew Ng and Pieter Abbeel [14], [1]. They developed a guidance system that uses Reinforcement Learning for low-level manipulation of the aircraft actuators to maintain a desired attitude, and they demonstrated their trained helicopter's capabilities in helicopter competitions requiring the aircraft to perform advanced aerobatic manoeuvres. Furthermore, we mention the noteworthy work of Jacopo Panerati, Hehui Zheng, and Angela P. Schoellig [15] for providing a compact, open-source and easy to use Gym-style environment for Quadcopter that supports the definition of multiple learning tasks (multi-agent and vision-based Reinforcement Learning) on a practical robotic application.

OpenAI GYM

OpenAI GYM is a toolkit for developing and comparing Reinforcement Learning algorithms. It makes no assumptions about the structure of the agent, and it is compatible with any numerical computation library such as TensorFlow, PyTorch or Theano. To be able to use the GYM API, it is recommended to have `python 3.5+` installed alongside `conda`. `conda` is an open-source package and environment management system that runs on Windows, macOS and Linux. It helps with installing, running and updating packages and their dependencies, and it creates and switches between fully isolated environment in a local development machine. After installing `conda`, we begin by creating a python virtual environment named "gym-pybullet-drones" by executing the command.

`conda create --name gym-pybullet-drones python=3.8.8` in the commandline. When the environment finishes creating, it should be activated to be used, for that, we execute this command: `source activate gym-pybullet-drones`.

Now, we should be able to install Git. Git is a free and open-source distributed software for

version control system designed to handle small and large project, offering remarkable speed, efficiency, and data-integrity. It allows software tracking changes in any set of files [5.2]. Git is usually used for coordinating work among developers and programmers around the world, collaboratively developing source code. It also supports distributed and non-linear workflows consisting of thousands of parallel branches running on different systems. To install Git, we need to type the following command to the commandline:

```
(gym-pybullet-drones) user@linux:~$ conda install -c anaconda git
```

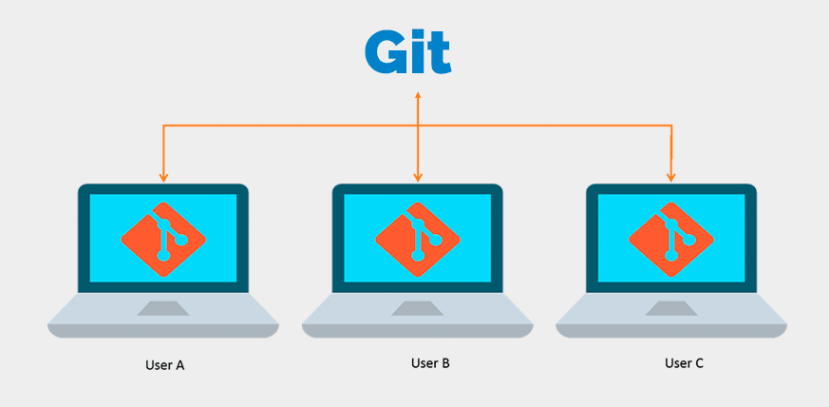


Figure 5.2: Git distributed workflow

Now we have Git installed, we can go ahead and clone the Github repository that we have created for this project, and install all the dependencies needed. To do this, we run the following commands:

```
(gym-pybullet-drones) user@linux:~$ git clone https://github.com/AbdelKariim
   /drone_pybullet
(gym-pybullet-drones) user@linux:~$ cd drone_pybullet
(gym-pybullet-drones) user@linux:~/drone_pybullet$ pip install -e .
```

An alternative way to get the code is by using Docker Containers. `docker` is a set of Platform as a Service (PaaS) product that uses OS-level virtualization to deliver software in packages called containers. Containers are isolated from one another and bundle their own software, libraries and configuration files. They can communicate with each other through well-defined channels. Because all of the containers share the services of a single Operating System Kernel, they use fewer resources than virtual machines. The figure [5.3] illustrates the working principle of docker virtualization. To install `docker` in the same isolated python environment as Git, we run the command: `conda install -c conda-forge/label/cf202003 docker`.

After succefully installing `docker`, we need to pull the Docker image that contains the source code that we have written. For that, we we have to execute the following commands:

```
(gym-pybullet-drones) user@linux:~$ docker push abdelkarim16/drone_pybullet:
   latest
(gym-pybullet-drones) user@linux:~$ docker run -it --name gym-pybullet-
   drones abdelkarim16/drone_pybullet:latest
root@1200adead522:/gym_pybullet_drones#
```

The shell presented to us indicates that we are successfully inside our Docker Container running as `root`. In this container, we already have `python` and `pip` running. Now, we install all the dependencies needed to run the project the same thing as we have done when using `git`, by using the command:

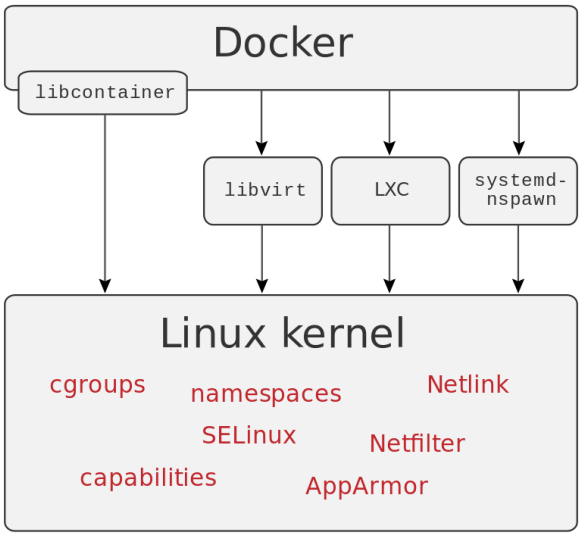


Figure 5.3: Docker Virtualization

```
root@1200adead522:/gym_pybullet_drones# pip install -e .
```

At this step, we have all the ingredients to successfully run our project without any errors.

PyBullet Physics Engine

PyBullet is an easy to use Python module for physics simulation or robotics, games, visual effects and Machine Learning. PyBullet gives the ability to load articulated bodies from URDF, SDF, MJCF, and other formats, it also provides forward dynamics simulation, inverse dynamics computation, forward and inverse kinematics, collision detection and ray intersection queries [5.4]. Aside from physics simulation, there are binding to rendering, with a CPU renderer such as TinyRenderer and OpenGL visualisation and support for Virtual Reality (VR) such as HTC Vive and Oculus Rift. PyBullet also has functionalities to perform collision detection queries like closest points, overlapping pairs and ray intersection test, and to add debug rendering (debug lines and text). PyBullet has cross-platform build-in client-server support for shared memory, UDP and TCP networking. So you can run PyBullet on Linux connecting to a Windows VR server. Aside from simplicity, PyBullet can be easily used with the most powerful Machine Learning framework like TensorFlow, PyTorch and OpenAI GYM. [3]

5.4 Simulation

The code downloaded in the last section comes with two GYM compatible environment for training our Reinforcement Learning agent, `TakeOff-Aviary` and `FlyThruGate Aviary` (for this project we will only use the first environment). These two environments can only work with single-agent Reinforcement Learning. The agent here is just the software running is Crazyflie 2.0 drone and the environment is everything else including the drone's dynamics. Each environment must implement the GYM interface and also contains all necessary functionalities to

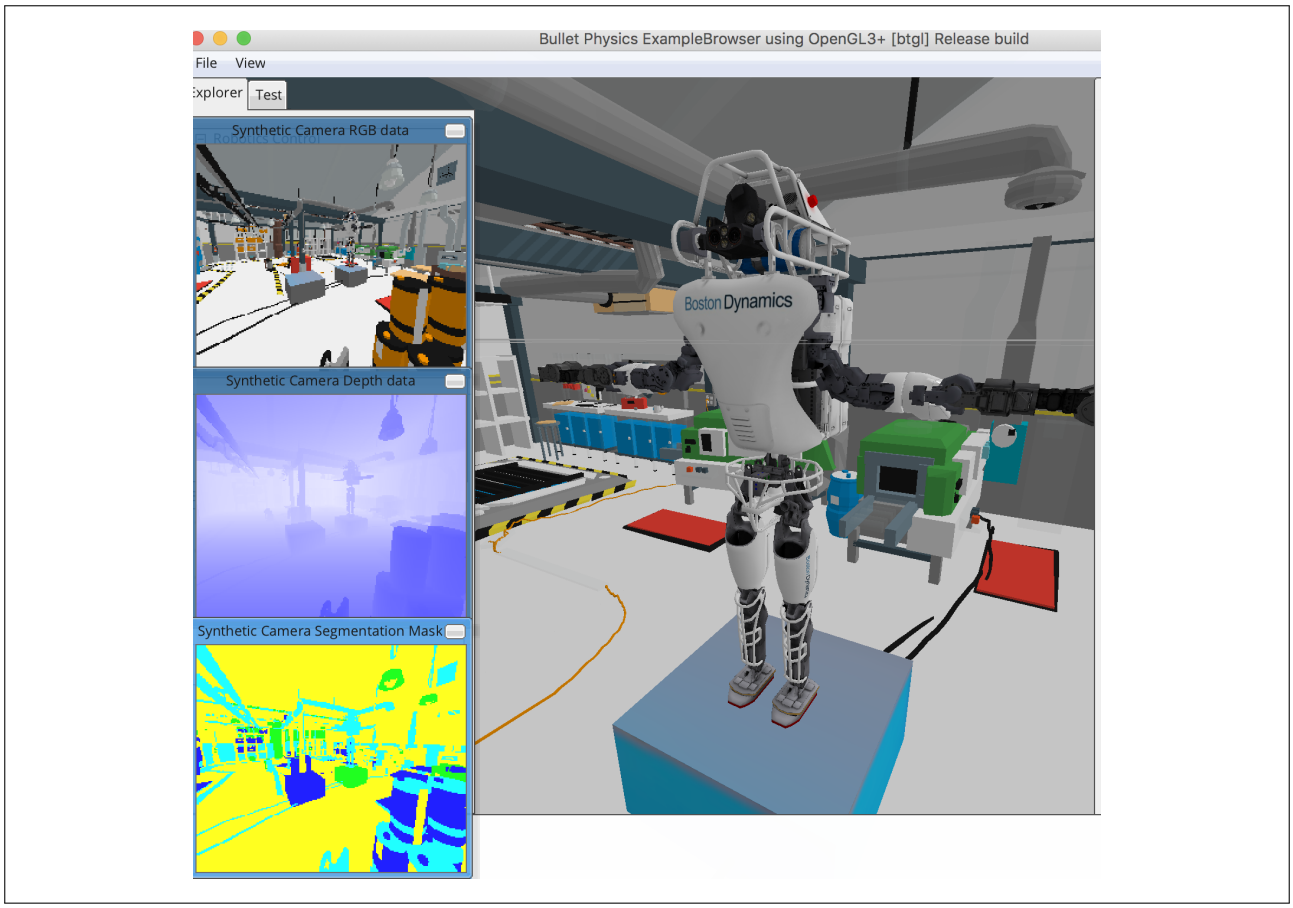


Figure 5.4: PyBullet Simulator

run the training agent and allow it to learn correctly. The following code illustrates the logic behind creating the Reinforcement Learning environments used to train the model:

```
import gym
class DroneEnv(gym.Env):
    metadata = {'render.modes': ['human']}
    def __init__(self):
            # Define action and observation space
            # They must be gym.spaces objects
            self.action_space = gym.spaces.Dict(0:Box(-1.0, 1.0,
               (4,), float32))
            self.observation_space = gym.spaces.Dict(0:Dict(
               neighbors:MultiBinary(1), state:Box(-inf, inf,
               (20,), float32)))
    def step(self, action):
            # Execute one time step within the environment
            """ Parameters
            ----------
            action : ndarray | dict[..]
                The input action for one or more drones,
                   translated into motor speed (RPMs)
```

```
            Returns
            -------
            ndarray | dict[..]
                The step's observations or state space """
            return state, reward, done, info
    def reset(self):
            # Reset the state of the environment to an initial
               state
            state = 0
            return state
    def render(self, mode='human', close=False):
            # Render the environment to the screen
```

In the class constructor, we begin by defining the type and the shape of the `action_space` which will contain all of the possible actions that could be taken by the training agent. The default action space used here is a dictionary whose keys are the drone indices and the values corresponding to the four normalized motor speeds in RPM. Similary, the `observation_space`, also called `state_space`, contains all of the environments data to be observed by the agent at every timestep.

The `reset` method will be periodically called to reset the environment to the initial observation. After that, the environment's `step` function after taking an action, will return the following four values:

- `observation` (object): the agent's current state or observation.
- `reward` (float): amount of reward achieved by the previous action. The goal obviously is to always increase the total cumulated reward. In `TakeOff-Aviary` the agent's goal is to reach a predetermined altitude and stabilize. The reward function is simply the negation of the squared Euclidean distance from the set point:

$$r = -\|[0, 0, 1] - \boldsymbol{x}\|_2^2 \tag{5.1}$$

- `done` (boolean): a flag whether to call the `reset` function. When `done == True`, it indicates that the learning episode has terminated
- `info` (dict): diagnostic information useful for debugging. It can sometimes be useful for learning. It might contain some raw probabilities behind the environment's last state change.

Finally, the `render` method may be called periodically to render the 3D drone environment.

5.4.1 Training

Writing vanilla Machine Learning models can be quite difficult task, due to insufficient data that can cause variance increase. High variability in models means that the model will perfectly fit the data it was given, but it ceases working as soon as new data is fed into. So powerful Machine Learning frameworks are then used for creating higher level of abstraction of

the core components of Reinforcement Learning algorithms, and to make code easier to develop and maintain. There are a variety of powerful Machine Learning frameworks, geared at different purposes. For our purpose, we will use **Stable Baselines** which is a set of high-quality implementations of the most popular Reinforcement Learning algorithms such as Advanced Actor-Critic (A2C), Proximal Policy Optimization (PPO), and Deep Deterministic Policy Gradient (DDPG), with a common interface based on **OpenAI Baselines** and built upon PyTorch library. This framework will be automatically installed in our project and ready to use. To be able to use custom GYM environments with the Stable Baselines3 framework, we need first to register the environment using the GYM registry API, then create a vectorized instance of the environment, and finally start the learning process.

```
from gym.envs.registration import register

register(
    id='takeoff-aviary-v0',
    # full relative path to environment code
    entry_point='gym_pybullet_drones.envs.single_agent_rl:
       TakeoffAviary',
)

env = gym.make('takeoff-aviary-v0')
check_env(env, warn=False, skip_render_check=True)
env = DummyVecEnv([lambda: env])
```

In this project, we used the default implementations of the Advanced Actor-Critic (A2C) algorithm provided by the Stable Baselines3 framework. We choose Multi-Layer Perceptron (MLP) models with ReLU activation function and 4 hidden layer, with 512, 512, 256, and 128 units respectively [15]. It should be noted that, in Reinforcement Learning algorithms, hyperparameters tuning is an omnipresent problem as it is an integral aspect of obtaining the state-of-the-art performance for any model. Most often, hyperparameters are optimized just by training a model on a grid of possible hyperparameters value by taking that one that performs the best on a grid search [8]. After a long process of trial-and-error and fine tuning the model's hyperparameters, we settled for values that gave the best results.

```
if args.algo == 'a2c':
    model = A2C(ActorCriticPolicy,
                env=env,
                learning_rate=0.0007,
                gamma=0.99,
                gae_lambda=1.0,
                ent_coef=0.0,
                vf_coef=0.5,
                rms_prop_eps=1e-5,
                use_rms_prop=True,
                use_sde=False,
                tensorboard_log=tensorboard_log,
                verbose=1
                )
```

```
reward_threshold = 0

callback_on_best = StopTrainingOnRewardThreshold(
   reward_threshold=reward_threshold,
                                verbose=1
                                )

eval_callback = EvalCallback(env,
                             callback_on_new_best=
                                callback_on_best,
                             verbose=1,
                             best_model_save_path=filename+'/
                                ',
                             log_path=filename+'/',
                             eval_freq=int(2000),
                             deterministic=True,
                             render=False
                             )

model.learn(total_timesteps=500000,
            callback=eval_callback,
            log_interval=100,
           )
```

In the code above, we used the Stable Baselines3 callbacks, which are a set of functions that will be called at given stages of the training procedure. We have primarily used to access internal state of the Reinforcement Learning model, they also help with monitoring, auto-saving, and model manipulation.

The `StopTrainingOnRewardThreshold`, as the name already suggests, is used to stop the training once a threshold in episodic reward has been reached (*i.e* when the model is good enough). In our case, the reward threshold is 0, because we defined the reward as the negation of the squared Euclidean distance from the set point. `EvalCallback` on the other hand, evaluates periodically the agent's performance, and saves the evaluations in a specified folder as numpy archive (evaluations.npz). We should note that, in the `EvalCallback`, we used `deterministic==True` to apply deterministic actions to the environments. After that, we trained the Reinforcement Learning agent for 500000 timesteps. Ideally, the number of timesteps should be very big as the Reinforcement Learning agent needs more timesteps to effectively learn how to fly the drone, but due to lack of powerful hardware and graphic computing resources, the training process was mediocre. This will further drastically improve when trained on powerful GPUs that have support for `CUDA` cores.

5.4.2 Training Results

After around 2 hours of training, it is time for visualizing the performance of the Reinforcement Learning agent, for this task, we used `TensorBoard` which is a visualization toolkit developed by TensorFlow team, and it is often used by researchers and engineers to visualize and understand Machine Learning experiments, share their research to help fine tuning models. Integrating

`TensorBoard` in our project helped us monitor hyperparameter and track our experiment metrics. The learning curves of our Reinforcement Learning model are presented below:

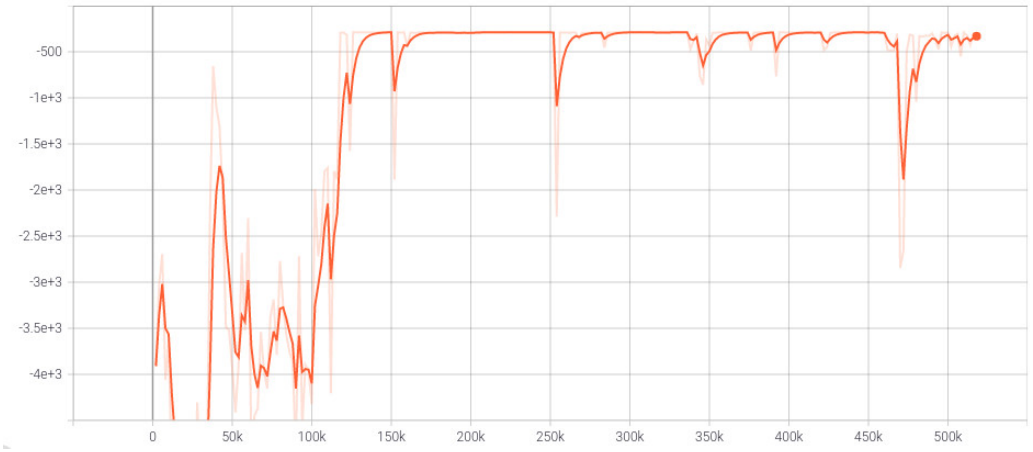


Figure 5.5: Timesteps mean reward

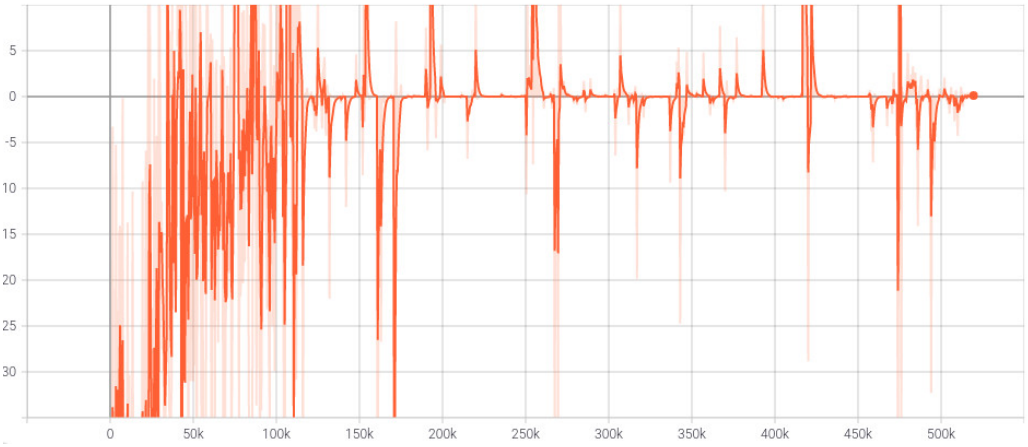


Figure 5.6: Policy Loss

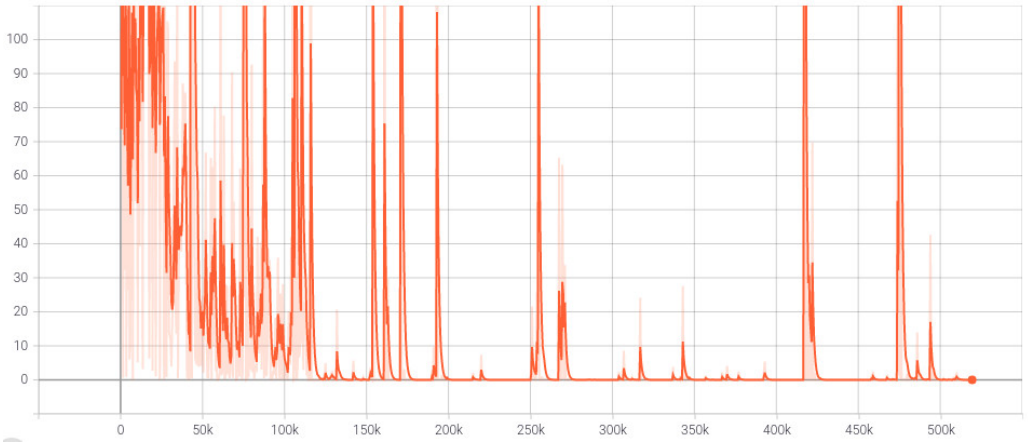


Figure 5.7: Value Loss

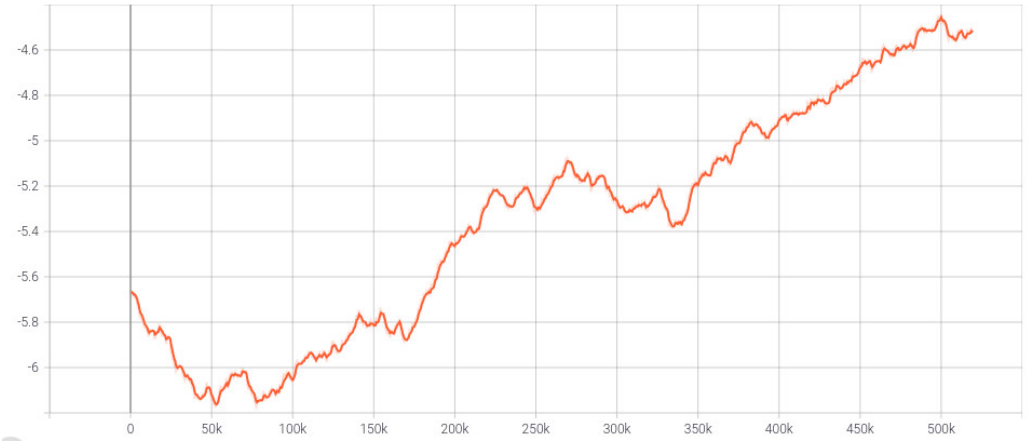


Figure 5.8: Entropy Loss

Figure 5.9: Standard deviation

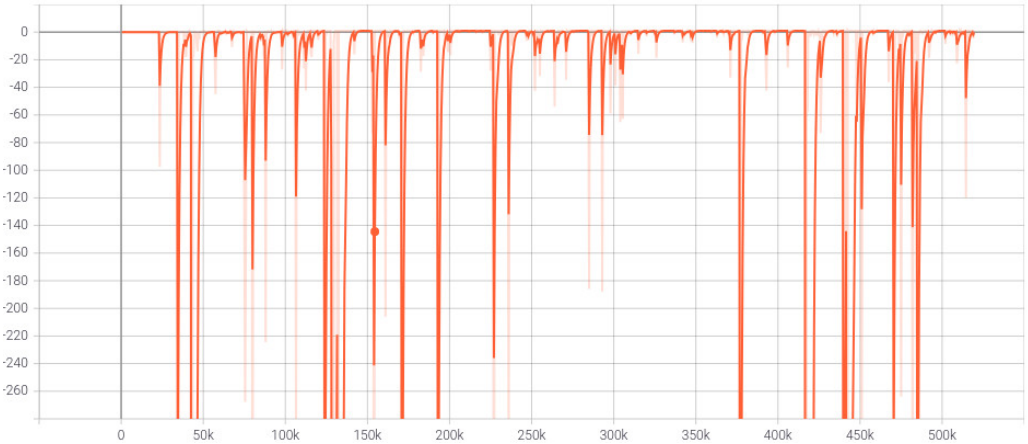


Figure 5.10: Explained Variance

The immediate observation that we can notice, is that 500000 timesteps was not sufficient for the agent. In the beginning of the learning process, the Entropy loss [5.8] was high as well as the standard deviation [5.9], which indicates that agent is having hard time choosing what actions to take, and it also indicated that there exist some randomness in the learning procedure. Moreover, the model's prediction of the Q-values estimate of each state is mediocre and it

correlates with the high mean magnitude of the policy loss function, [5.6], [5.7]. Consequently, the total cumulative reward gained by the agent is pretty low. After about 150000 time steps, the agent starts to show some remarkable improvement as the mean reward settled around -200 [5.5] still far from the threshold but acceptable, the Entropy loss increases to zero, the standard deviation becomes much smaller than before, and the explained variance decreases to zero, and the overall performance shows some significant improvement resulting in a successful fly.

5.4.3 Testing Results

After the Reinforcement Learning agent finished training, it was time for testing. For this, we loaded the saved pre-trained model and tested it on the CrazyFlie drone. Despite the relatively low mean reward gained by the training agent and the limited hardware resources, the agent successfully took-off the drone and hover it 1m above the ground exactly as what he was trained for [5.14]. From figures 5.11, 5.12, 5.13, we notice that the quadcopter was steady and stable at maintaining the altitude despite some tiny x and y deviations. The roll and pitch angles changed a bit before reaching the desired altitude due to the coupling between the translation and rotation dynamics of the quadcopter 5.12. The motors speed 5.13 was high at around 20000 revolution per minute (rpm) when taking-off, but quickly droped to 15000 rpm in steady state. Without any knowledge about the quadcopter dynamics, the Reinforcement Learning agent showed some robustness and a certain adaptation for the environment to accomplish its objective. The good aspect of our code is it re-usability. You can train the same Reinforcement Learning agent for different task that he has never seen before by just shaping the reward signal, train it on a good hardware to achieve best results, and then test it in different dynamic environment to see if the training process was effective or not. This rather simple application opens new perspectives to quadcopter attitude control that would be very interesting to test such as trajectory planning, obstacle avoidance and autonomous navigation.

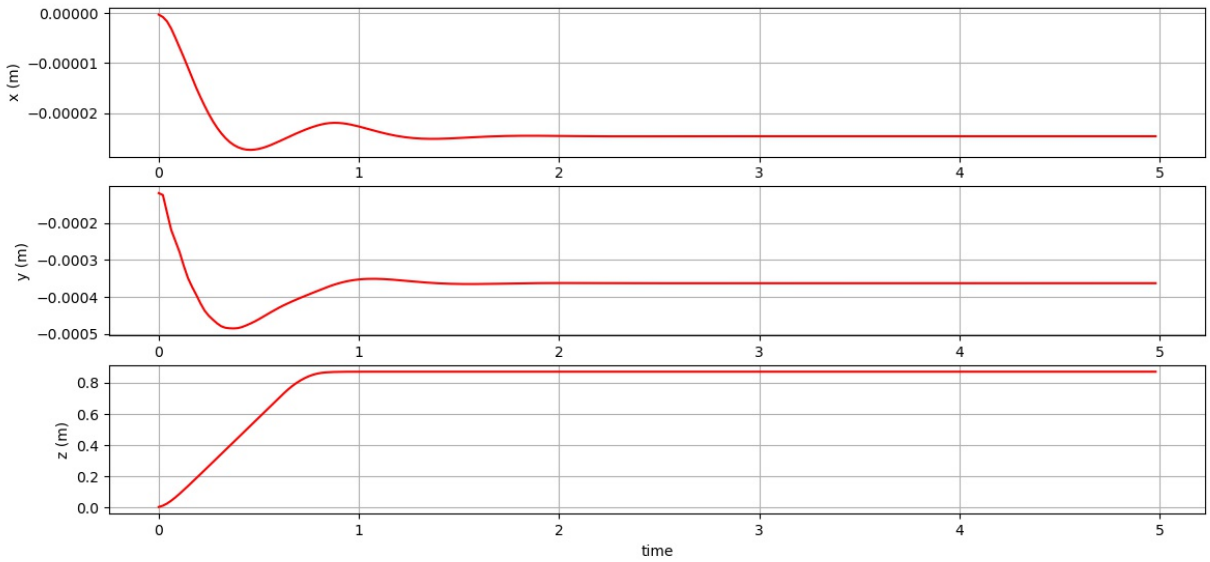


Figure 5.11: Drone position

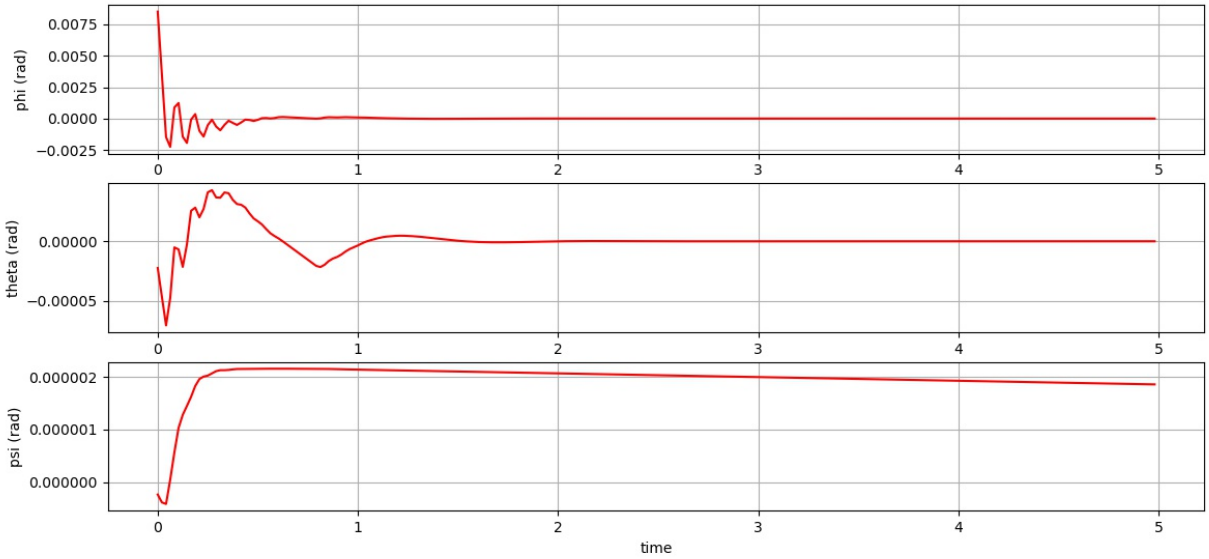


Figure 5.12: Drone orientation

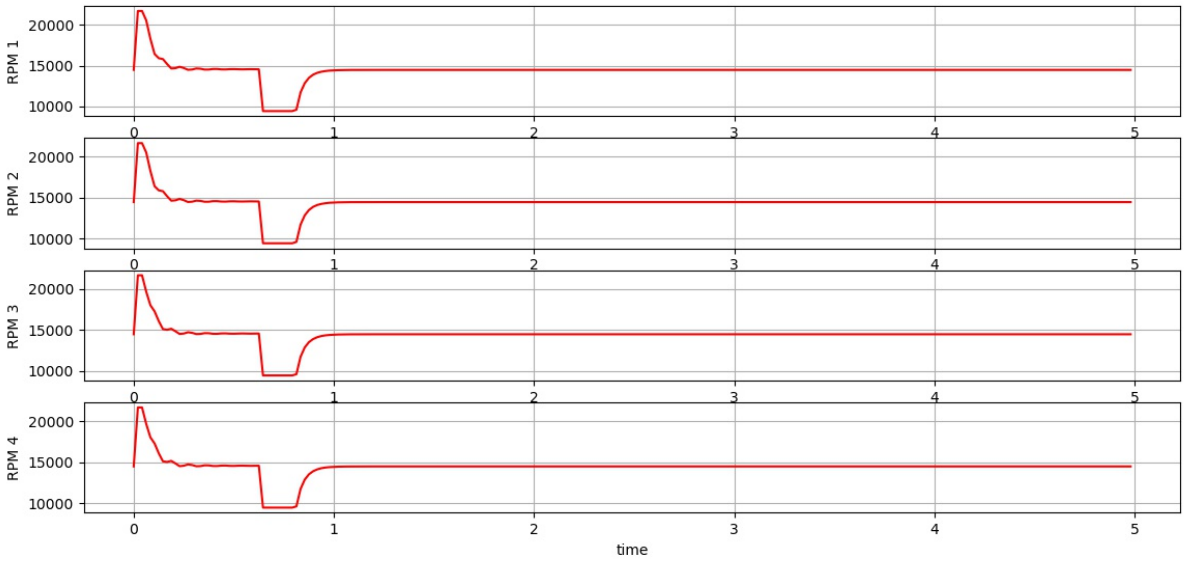


Figure 5.13: Motors speed

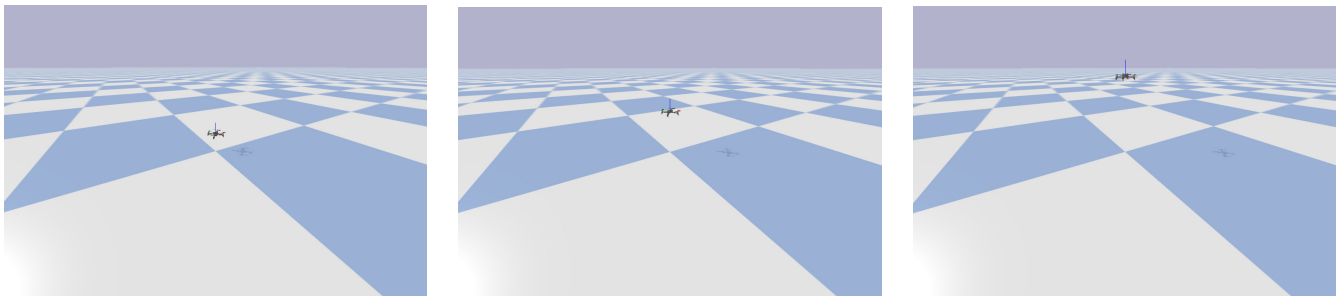


Figure 5.14: Quadcopter Simulation in PyBullet

5.5 Future Work and Conclusion

In this chapter, we presented the Reinforcement Learning training environment built upon the PyBullet physics engine, and the OpenAI GYM API and Stable Baselines framework to evaluate performance of state-of-the-art Advanced Actor-Critic algorithm to synthesize and develop intelligent and robust attitude controller for drones. The results we obtained highlight that our Reinforcement Learning agent was unexpectedly good at trajectory tracking when used in single-agent testing, and it performed exceptionally well also in continuous tasks without retraining. Despite this remarkable success, we rather consider this project results as a first milestone and a good motivation to further inspect the boundaries of Reinforcement Learning for intelligent control, and help demystify the difficulties found when developing software solutions for robotics [10] . We plan to develop a new reward engineering to harness the true power of Reinforcement Learning ability to adapt and learn in more complex and dynamic environments (ground effect, drag effect, and downwash), emphasise on digital twinning concept to allow easy transferability to real hardware [10] .

CONCLUSION

Unmanned Aerial Vehicle navigation and control have been a developing topic due to its vast range of applications. UAVs are employed for a variety of purposes, including civilian tasks, military missions, object tracking, and search and rescue operations. Each of these applications requires accurate navigation in order to avoid collisions while taking the most efficient path to the destination.

We began our thesis by introducing the concept of Reinforcement Learning as a computational approach to understanding and automating decision making and direct goal learning. We provided a clear and simple account of the key ideas and algorithms of Reinforcement Learning, and its simplest aspects and main distinguishing features. We covered also the connections between Reinforcement Learning and Optimal Control from control theory perspective. Later on, we introduced the framework of Markov Decision Processes as a way to enable interaction between the learning agent and its environment to achieve better goals. Also, we developed the concept of value function defined as an agent's ability of learning to act optimally by observing the consequences of his actions. We also examined an alternative to value iteration which is policy iteration. Policy iteration helps the agent learn a parameterized policy that allows actions to be conducted without checking action-value estimations.

In the last chapter, we trained a Reinforcement Learning agent in a training environment build upon PyBullet and OpenAI library. The Advanced Actor Critic A2C algorithm showed promising results in continuous tasks effectively maintaining the drone at a fixed altitude without any prior knowledge of the quadcopter dynamics. The training results were limited by the computational constraints of the hardware in which we ran the simulation. To obtain better results, it is recommended to use online cloud computing services such as Google Cloud, Amazon AWS or Linod, which can provide cutting edge hardware resources for Machine Learning and Artifitical Intelligence.

CHAPTER

6

ANNEXE

6.1 Nonlinear Stability

A system is called asymptotically stable around it's equilibrium point if it satisfies the following two conditions:

- $\forall \epsilon > 0$, $\exists \delta_1 > 0$ such that if $\|x(t_0)\| < \delta_1 \Rightarrow \|x(t)\| < \epsilon$
- $\exists \delta_2$ such that if $\|x(t_0)\| < \delta_2$ then, $\lim_{t \to +\infty} x(t) = 0$

6.1.1 Lyapunov Direct Method

Let's consider the following unforced continuous-time system $\dot{x} = f(x(t))$ with $\bar{x} = 0$ an equilibrium point, the idea behind Lyapunov direct method is to establish propreties of the equilibrium point by studying how certain carefully selected scalar functions ($\mathcal{V}$) of the state behaves as the system state evolves. Let $\mathcal{V}$ be a continuous map from $\mathbb{R}^N \Rightarrow \mathbb{R}$, we call $\mathcal{V}(x)$ locally positive definite (lpd) around $x = 0$ if:

- $\mathcal{V}(0) = 0$
- $\mathcal{V}(x) > 0$ for $0 < \|x\| < r$ for some r

We shall consider $\mathcal{V}(x)$ that have continuous first partial derivative *i.e*: $\mathcal{V} \in \mathcal{C}^1$ and let $\dot{\mathcal{V}}(x)$ be the time derivative of $\mathcal{V}(x)$, then:

$$\begin{aligned} \dot{\mathcal{V}}(x) &= \frac{d\mathcal{V}(x)}{dt} = \langle \nabla \mathcal{V}(x), \dot{x} \rangle \\ &= \frac{d\mathcal{V}(x)}{dt}.\dot{x} = \frac{d\mathcal{V}(x)}{dt}.f(x(t)) \end{aligned}$$

for $\mathcal{V}$ to be a Lyapunov function we must have $\dot{\mathcal{V}}$ locally negative semi-definite.

6.1.2 Lyapunov theorem for local stability

If there exist a Lyapunov function of a system described by the equation $\dot{x} = f(x(t))$ then $\bar{x} = 0$ is a stable equilibrium point in the sense of Lyapunov. In addition $\dot{\mathcal{V}}(x) < 0$ for $0 < \|x\| < r$ then $\bar{x} = 0$ is an asymptotically stable equilibrium point.

6.1.3 Lyapunov theorem for global asymptotic stability

if $\mathcal{V}(x)$ is positive definite on the entire state space and $|\mathcal{V}(x)| \to \infty$ as $\|x(t)\| \to \infty$ and it's derivative $\dot{\mathcal{V}}$ is negative definite on the entire state space, then the equilibrium point is glabally asymptotically stable.

6.1.4 The indirect method of Lyapunov

The indirect method of Lyapunov uses the linearization of a system to determine the local stability of the original system. Let's Consider $\dot{x} = f(x(t), t)$ with $f(0, t) = 0\ \forall t$ then:

$$A(t) = \frac{\partial f(x,t)}{\partial x}\Big|_{x=0} \tag{6.1}$$

for an uniform linearization of the system, we require the strong condition such that:

$$\lim_{x\to 0} \sup_{t\geq 0} \frac{\|f(x,t)\|}{\|x\|} = 0 \tag{6.2}$$

For a linear system described by the equation $\dot{z} = A(t)z$ to be asymptotically stable we need the eigenvalues of A to be in the open left half plan (LHP) *i.e* $\mathrm{Re}(\lambda_i) < 0$

6.2 BackStepping Control Strategy

Let's consider the following nonlinear system:

$$\begin{cases} \dot{x}_1 = f_1(x_1) + g_1(x_1)x_2 \\ \dot{x}_2 = f_2(x_1, x_2) + g_2(x_1, x_2)x_3 \\ \vdots \\ \dot{x}_{n-1} = f_{n-1}(x_1, x_2, ..., x_{n-1}) + g_{n-1}(x_1, x_2, ..., x_{n-1})x_n \\ \dot{x}_n = f_n(x_1, x_2, ..., x_n) + g_n(x_1, x_2, ..., x_n)u \end{cases} \tag{6.3}$$

and u is the control input
Let's assume we have:

$$\begin{cases} \dot{\eta} = f(\eta) + g(\eta)\sigma \\ \dot{\sigma} = u \end{cases} \tag{6.4}$$

$[\eta\ \sigma]^T$ is the state vector $\in \mathbb{R}^3$ and u is the control input, suppose that σ is a virtual control input such that

$$\sigma = \phi(\eta) \tag{6.5}$$

substituting (2) in (1) we obtain:

$$\dot{\eta} = f(\eta) + g(\eta)\phi(\eta) \tag{6.6}$$

to demonstrate the asymptotic stability of the origin of (3), we have to had a scalar function $\mathcal{V}$ such that:

- $\mathcal{V}(\eta)$ is positive definite and $\mathcal{V}(0) = 0$
- $\dot{\mathcal{V}}$ is semi-negative definite:

$$\begin{aligned} \dot{\mathcal{V}}(\eta) &= \frac{d\mathcal{V}(\eta)}{dt} \\ &= \frac{d\mathcal{V}(\eta)}{d\eta}\frac{d\eta}{dt} = \frac{d\mathcal{V}(\eta)}{d\eta}\dot{\eta} \\ &= \frac{d\mathcal{V}(\eta)}{d\eta}[f(\eta) + g(\eta)\phi(\eta)] \leq -\mathcal{W}(\eta) \end{aligned}$$

So starting from $\dot{\eta} = f(\eta) + g(\eta)\sigma$ adding and subtracting $g(\eta)\phi(\eta)$ we obtain:

$$\begin{aligned} \dot{\eta} &= f(\eta) + g(\eta)\sigma - g(\eta)\phi(\eta) + g(\eta)\phi(\eta) \\ &= f(\eta) + g(\eta)\phi(\eta) + g(\eta)(\sigma - \phi(\eta)) \end{aligned}$$

with: $\sigma = u$ from (38).
Let denote by $e_\sigma = \sigma - \phi(\eta)$ as the error between the state and the virtual input, or goal here to force σ to be equal to $\phi(\eta)$. Writing the state equations in terms of e_σ:

$$\begin{cases} \dot{\eta} = f(\eta) + g(\eta)\phi(\eta) + g(\eta)e_\sigma \\ \dot{e_\sigma} = u - \dot{\phi(\eta)} \end{cases} \tag{6.7}$$

where:

$$\begin{aligned}\dot{\phi(\eta)} &= \frac{d\phi(\eta)}{dt} = \frac{d\phi(\eta)}{d\eta}\dot{\eta} \\ &= \frac{d\phi(\eta)}{d\eta}[f(\eta) - g(\eta)\sigma]\end{aligned}$$

Setting $u = v + \dot{\phi(\eta)}$:

$$\begin{cases}\dot{\eta} = f(\eta) + g(\eta)\phi(\eta) + g(\eta)e_\sigma \\ \dot{e_\sigma} = v\end{cases} \tag{6.8}$$

Similar to (48), the Lyapunov condidate function can be taken:

$$\mathcal{V}_c(\eta, \sigma) = \mathcal{V} + \frac{1}{2}e_\sigma^2 \tag{6.9}$$

so:

$$\begin{aligned}\dot{\mathcal{V}}_c(\eta, \sigma) &= \frac{\partial \mathcal{V}(\eta)}{\partial \eta}[f(\eta) + g(\eta)\phi(\eta)] + \frac{\partial \mathcal{V}(\eta)}{\partial \eta}g(\eta)e_\sigma + e_\sigma\dot{e_\sigma} \\ \dot{\mathcal{V}}_c(\eta, \sigma) &\leq -\mathcal{W}(\eta) + \frac{\partial \mathcal{V}(\eta)}{\partial \eta}g(\eta)e_\sigma + e_\sigma v \qquad \text{(from (51))}\end{aligned}$$

We take: $v = -\dfrac{\partial \mathcal{V}(\eta)}{\partial \eta}g(\eta) - ke_\sigma$ for $k > 0$, substituting v in the above equation, we obtain:

$$\dot{\mathcal{V}}_c \leq -\mathcal{W}(\eta) - ke_\sigma^2 \tag{6.10}$$

Which shows that the origin ($\sigma = 0$ and $\eta = 0$) is asymptotically stable. The control input to the original system u is:

$$\begin{aligned}u &= -\frac{\partial \mathcal{V}(\eta)}{\partial \eta}g(\eta) - ke_\sigma + \frac{d\phi(\eta)}{d\eta}[f(\eta) - g(\eta)\sigma] \\ &= \frac{d\phi(\eta)}{d\eta}[f(\eta) - g(\eta)\sigma] - \frac{\partial \mathcal{V}(\eta)}{\partial \eta}g(\eta) - k[\sigma - \phi(\eta)]\end{aligned}$$

BIBLIOGRAPHY

[1] Abbeel, P., Coates, A., Quigley, M., and Ng, A. Y. An application of reinforcement learning to aerobatic helicopter flight. *Advances in neural information processing systems 19* (2007), 1.

[2] Atienza, R. *Advanced deep learning with Keras: apply deep learning techniques, autoencoders, GANs, variational autoencoders, deep reinforcement learning, policy gradients, and more*. Packt Publishing, 2018.

[3] Coumans, E., and Bai, Y. Pybullet, a python module for physics simulation for games, robotics and machine learning. http://pybullet.org, 2016–2021.

[4] Dierks, T., and Jagannathan, S. Output feedback control of a quadrotor uav using neural networks. *IEEE transactions on neural networks 21*, 1 (2009), 50–66.

[5] Halgasik, J. *Flight Control System Unit for Small UAV Aircraft*. PhD thesis, Diploma thesis. CTU Prague, 2014.

[6] Hassanalian, M., and Abdelkefi, A. Classifications, applications, and design challenges of drones: A review. *Progress in Aerospace Sciences 91* (2017), 99–131.

[7] Hasselt, H. Double q-learning. *Advances in neural information processing systems 23* (2010), 2613–2621.

[8] Jomaa, H. S., Grabocka, J., and Schmidt-Thieme, L. Hyp-rl: Hyperparameter optimization by reinforcement learning. *arXiv preprint arXiv:1906.11527* (2019).

[9] Joshinav, Oct 2013.

[10] Koch, W., Mancuso, R., West, R., and Bestavros, A. Reinforcement learning for uav attitude control. *ACM Transactions on Cyber-Physical Systems 3*, 2 (2019), 1–21.

[11] Konda, V. R., and Tsitsiklis, J. N. Actor-critic algorithms. In *Advances in neural information processing systems* (2000), Citeseer, pp. 1008–1014.

[12] Lapan, M. *Deep reinforcement learning hands-on: apply modern RL methods to practical problems of chatbots, robotics, discrete optimization, web automation, and more*. Packt Publishing, 2020.

[13] Moghadam, P. H. Deep reinforcement learning: Dqn, double dqn, dueling dqn, noisy dqn and dqn with prioritized..., Jul 2019.

[14] Ng, A. Y., Kim, H. J., Jordan, M. I., Sastry, S., and Ballianda, S. Autonomous helicopter flight via reinforcement learning. In *NIPS* (2003), vol. 16, Citeseer.

[15] Panerati, J., Zheng, H., Zhou, S., Xu, J., Prorok, A., and Schoellig, A. P. Learn-

ing to fly—a gym environment with pybullet physics for reinforcement learning of multi-agent quadcopter control, 2021.

[16] Sutton, R. S., Bach, F., and Barto, A. G. *Reinforcement Learning: An Introduction*. MIT Press Ltd, 2018.

[17] Valavanis, K. P. *Advances in unmanned aerial vehicles: state of the art and the road to autonomy*. Springer, 2010.

[18] Wang, H., Zariphopoulou, T., and Zhou, X. Y. Exploration versus exploitation in reinforcement learning: a stochastic control approach. *Available at SSRN 3316387* (2019).

[19] Watkins, C. J., and Dayan, P. Q-learning. *Machine learning 8*, 3-4 (1992), 279–292.

Abstract

In this work, we provided an introduction to Reinforcement Learning concepts and paradigms and their growing importance in the robotics research community. We proposed an open-source OpenAI GYM-like environment for training a Reinforcement Learning agent to fly a quadcopter based on the PyBullet physics engine. This type of environment coupled with a problem specification of a reward function is important to standardize the development and the benchmarking of learning algorithms and help combine the advantages of using Control Theory alongside Machine Learning.

Résumé

Dans ce travail, nous avons présenté les concepts et paradigmes de l'apprentissage par renforcement et leur importance croissante dans la communauté de recherche en robotique. Nous avons proposé un environnement open-source de type OpenAI GYM pour entraîner un agent d'apprentissage par renforcement à piloter un quadcoptère basé sur le moteur physique PyBullet. Ce type d'environnement, associé à la spécification d'une fonction de récompense, est important pour normaliser le développement et l'évaluation des algorithmes d'apprentissage et pour aider à combiner les avantages de l'utilisation de la théorie du contrôle et de l'apprentissage automatique.

تلخيص :

في هذا العمل، قدمنا مقدمة لمفاهيم ونماذج التعلم المعزز وأهميتها المتزايدة في مجتمع أبحاث الروبوتات. لقد اقترحنا بيئة مفتوحة المصدر شبيهة ببرنامج OpenAI GYM لتدريب وكيل التعلم المعزز على قيادة طائرة بدون طياراستنادا إلى المحرك الفيزيائي PyBullet. يعد هذا النوع من البيئة مقترنا بمواصفات مشكلة لتوظيف المكافأة أمرا مهما لتوحيد التطوير والقياس المعياري لخوارزميات التعلم والمساعدة في الجمع بين مزايا استخدام نظرية التحكم جنبا إلى جنب مع التعلم الآلي.

Key Words

Reinforcement Learning, Machine Learning, PyBullet, Stable BaseLines, OpenAI GYM, Quadcopter, Attitude Control.