UNIVERSIDAD DE LAS AMÉRICAS PUEBLA

ESCUELA DE INGENIERÍA

DEPARTAMENTO DE COMPUTACIÓN, ELECTRÓNICA Y MECATRÓNICA

Fault Tolerant Control Applied to Aerial Vehicles

Tesis que, para completar los requisitos del Programa de Honores presenta el estudiante

Sergio Noriega Heredia

ID 153222

Ingeniería Mecatrónica

Dr. César Martínez Torres

San Andrés Cholula, Puebla. PRIMAVERA 2020

Hoja de firmas

Tesis que, para completar los requisitos del Programa de Honores presenta el estudiante **Sergio Noriega Heredia ID 153222**.

Director de Tesis

Dr. César Martínez Torres

Presidente de Tesis

Dr. José Luis Vázquez González

Secretario de Tesis

Dr. Gibran Etcheverry Doger

To my family, to my mentors and to my friends which made this possible.

Thank you so much for inspiring me to learn and grow.

Contents

1 Introduction **2**

2 Background Knowledge **2**
2.1 Drone . . . 3
2.1.1 UAV . . . 6
2.1.2 Quadcopter . . . 8
2.1.3 Flight Principle . . . 11
2.1.4 Normativity . . . 15
2.2 Simulator . . . 16
2.2.1 ROS . . . 16
2.2.2 Gazebo . . . 18
2.3 Control Theory . . . 19
2.3.1 Fault Detection . . . 21
2.3.2 Fault Tolerant Control . . . 23

3 Project Description **24**
3.1 Elements . . . 24
3.2 Control Models . . . 26

4 Experimental Results **30**

5 Conclusions **33**

6 References **34**

1 Introduction

Drones have become more relevant since their wide commercial introduction about 10 years ago, in 2010. Their popularity rose as many companies started selling them as expensive toys. And with that, a whole community of enthusiasts and engineers appeared to give them their place in our parks and testing facilities. Nowadays, the word *drone* is now a widely adopted term. It has become part of everyday speech, just as many other technological terms in the past 50 years. We usually hear it in media broadcasts, magazines, and everyday conversations. However, there is a much longer story behind them.

Drones have been object of both development hype and public rage throughout the years. Both have been very valid points of view. This is because they are useful as they can be dangerous. Initially, people started using them in a variety of manners. Sometimes we hear about doing photography with a drone, or explore the skies. Other times, it is related to military applications, and for defense and rescue operations. There are even competitions based on drone racing, with many professional pilots fighting to finish in first place. However, some of the most impacting news and videos related to drones have been about damage and harm done by them. I think that is why, in recent days, they have taken great backslash from the general public.

Nevertheless, they are much more than expensive dangerous toys. I believe drones have a much wider potential, and may be the tools of the future. They can move fast, be efficient, carry stuff... with just that, their usefulness as a vehicle is potentially more than any traditional car. My posture is we just need to develop the correct safety measures and tools to set the base for a new drone era. They are still in development and refinement, so there is still more ground we can cover to make them better. And in this work, I will attempt to do that.

This thesis covers an engineering project aiming to enhance and test drone capabilities. The main focus is on a commercial drone. A set of new functions are added to it, and its performance is tested after the enhancement. There is a wide area in which we can act upon, and this work tackles one of the common problems present. Failures are not as uncommon in drones, making it a reason why they are considered dangerous. This work provides a solution to that problem, by acknowledging when it happens and then compensate to suppress the fault.

2 Background Knowledge

In order to comprehend the concepts presented deeper into in this work, we must first define them generally. This section starts with general definitions for drones up until how quadcopters, a type of drone, achieve movement.

2.1 Drone

As used in the first section, the term *drone* could be replaced as any flying device, like the ones that we see from time to time in the skies or maybe, more specifically, at a concert. Nonetheless, the word has a much wider definition, far more general than the public recognizes it to be. A *drone* is any kind of vehicle which can move by itself and has no human driving it onboard. In other words, a drone is *autonomous* and *unmanned*.

But, what about the flying part? There is where the broader meaning takes part. A drone is not necessarily of the flying type. A drone can do its primary activities in all known environments. For air, land, water, and even space drones, we have an specific category, or if you wish to call it, a technical name to categorize them. They can also be used for many kinds of activities, which will be briefly described for each category.

First of all, an Unmanned Ground Vehicles (UGV) are the ones which work on land. These either move on treads or wheels, and are usually employed in situations where it is too dangerous or risky for humans to work in. Examples for these are remote bomb disarming operations, radioactive environment operations, firefighting, search and rescue operations, among many others.

Figure 1: Example of an experimental UGV. [Väljaots and Sell, 2019]

Other category is for Unmanned Surface Vehicles (USG), which in turn, are drones which move on the surface of a liquid. They are principally found at sea for scientific research and data collection. Originally, they were used for surveillance and mine sweeping, which were military applications. However, in most recent years, unmanned barges or Autonomous Spaceport Drone Ship (ASDS) were invented for oceanic landings of SpaceX rockets. [Bergin, 2014]

Another category is Unmanned Underwater Vehicles (UUV). These drones submerge themselves in water and engage in conditions where either it is too expensive or too dangerous for humans to go. Most applications of these are data gathering, mine sweeping, maintenance operations, and underwater exploration.

Unmanned Spacecrafts, or "space drones", fall into a special category because most, if not all uncrewed spacecrafts are vehicles carrying a payload. So, to a certain extent,

Figure 2: Example of a deployed USG. [SpaceX, 2020]

Figure 3: Example of an experimental UUV. [Olejnik, 2016]

all probes out there are drones that operate in high atmosphere or in space. This definition includes all satellites and remotely controlled spacecrafts. Telecommunications satellites, space telescopes, surveillance probes, and many other fall into this category. Their main means of movement is by some kind of propulsion system, the most common one being material expulsion. However, most recently developed unmanned space probes are used to send resources to and from the International Space Station. Other recent applications include self-guided rocket boosters, and the new nano-satellite concepts.

As a final group, there are Unmanned Aerial Vehicles (UAV), which are the flying type drones. These have ability to take off and fly in air or a gas for an extended amount of time. Most commercially available kits fall into this category. Their applications include but are not limited to: photography, exploration, scientific research, surveillance, disaster response, data collection, combat, delivery, defense, automation, etc. UAVs are the main focus of this work, and thus, are later described in detail.

As it is demonstrated, a drone is not only a flying machine. It can be of many different types, and is used and built for very specialized tasks. As a result, they prove to be handy and unique tools for the job.

Figure 4: Example of an unmanned spacecraft in a mission. [SpaceX, 2020]

Figure 5: An example of a racing UAV. [Kaufmann et al., 2019]

Another common assumption is that all drones are robots, and that is not true either. And the reason for that is simple. A robot is an autonomous system which has the ability to process information coming from its sensors and take decisions by itself based on that information received. It is true that some drones have those functionalities (robot-like), but not all of them can do that. So, drones can be autonomous, and have advanced automated control implemented into it; or can also be remotely guided and controlled by a human from a remote location (no robotic functions).

This comparison can also be seen from a control theory standpoint. It is referred as the *loop type* of the system, and is explained as the type of control loop implemented. An open loop consists of a system with inputs, where a reference is provided but the output is not evaluated for input correction. In simpler terms, an open loop can be viewed as "eyeballing" the result and just hoping the output is the one desired. A closed loop is a system which is capable of achieving certain output state based on a reference, introducing feedback from the output and comparing it against the input. Robotics is based on closed loop systems, and is how basic control is implemented. However, control theory is a topic deeply discussed later on. Moreover, what we can salvage from this sections is that robotic type UAVs are the ones we are interested in, because of its

control capabilities and broader application use in the real world.

2.1.1 UAV

UAVs have a longer history than most people may recognize [O'Donnell, 2019] [Digital Trends, 2018]. First concepts were used in war in the 19th century. Both Americans and Austrians claim to have invented this device using balloons and explosives, to deliver remotely dropped explosives. They were not really as we conceive modern UAVs, but there were already unmanned flying devices. Later, in The Great War, recognizance flights were done to take photographs using the first winged unmanned aircrafts. Afterwards, in the 1930s, development for remotely controlled machines was executed, resulting in first remotely controlled airplanes.The Second World War pushed further development for these machines, and mass production for these airplanes could start. The later decades continued development on drones, principally as military toys. It was not until the 1960s, when first hobbyists started using them as toys,thanks to advancements in electronics and materials science. Military R&D kept making them more reliable until 1982, when Israeli UAVs won a fight over Syrian forces and demonstrated UAV feasibility as the new weapons in the sky. In 1986, a new drone, Pioneer was created as a recognizance drone by USA and Israel. Eventually, in the 1990s, technology got smaller enough to create mini and micro scale models of drones. A very famous use of drones in the early 2000s was for the "Anti-terrorist" effort lead by USA. Finally, first models of commercial quadcopters started appearing in 2010 with the introduction of the Parrot. Briefly after, new technology firms entered the market and started exploring the possibilities for them. Some innovated in logistics, like Amazon with deliveries; and some other innovated in product satisfaction and marketing, like DJI with its Phantom drones. The truth is that today, we live in a world where drones are widely accepted, and are also respected for their capabilities and applications.

So, after looking at the preceding definitions, now we know a UAV is a type of drone which is capable of taking off and flying. It is a neat definition, but is not enough to be specific. There are other characteristics we must establish in order to fully understand how it works.

First of all, a UAV has a method of flying. In this planet there are several ways to achieve this, based on different lift types: rotary propeller (rotary wing), fixed wing, flapping wing, propellant, and buoyant. Each type provides advantages and disadvantages while in flight, and must be carefully selected for each application. The one chosen for this work, and the most widely used, is the one which uses rotary propellers as its propulsion method. There are a few motives for its popularity and general superiority over the other models.

Since rotary propellers operate with electrical motors, fuel is cheaper and more accessible comparing it to a propellant based build. Electricity is cheap to acquire and can be found just reaching to an electrical outlet. In contrast, gasoline or pressured gases are more difficult to obtain, and increase relative risk because of combustion engines or pressurized tanks onboard. Now, compared to a fixed and flapping winged UAV, a

propeller UAV is more maneuverable. Fixed winged devices must have some sort of run (either on a runway or catapulted) before wings generate lift, whilst a propeller UAV can start flying in the same place where it starts. Following that premise, environmental requirements for flight are easier to achieve with propeller UAVs, making them dynamic and ideal for most kinds of environment. On the other hand, control for flapping winged UAVs is more complicated and the least explored. Lastly, blimps generate lift based on gas containers attached to their structure. This makes them the most efficient talking about energy efficiency, but tend to be bigger, less maneuverable, and cannot carry much weight.

Still, there is a tradeoff between using propeller and other kinds of lift. A propeller UAV will fly for a shorter amount of time compared to blimps or winged devices with the same amount of energy. This is due to the greater energetic demand from the greater count of motors. Fixed wing UAVs and blimps need only a pair of small motors for movement, while the most common kind of propeller UAVs have four and are comparatively more powerful.

Expanding more on motors, a UAV can also vary in composition regarding number and configuration of its lift generators. This does not influence in how they produce lift, but rather which are its specific capabilities. Some propeller UAVs have a different amount of propellers than a standard 4 propeller *quadcopter* and may vary its positioning depending on its base design. Common propeller configurations are coaxial (one on top of other), tandem/symmetric (same plane), and orthogonal (perpendicular to each other). Frequently, control is more easily achievable with an even number of propellers arranged in a symmetric fashion, but other options exist. The most remarkables are the 3 propeller planar, and the 2 propeller coaxial arrangements. Nonetheless, the most used one is yet the 4 propeller configuration in either "X" or "+" configuration, because it is the cheapest, most stable configuration.

Figure 6: A virtual model of a quadcopter. [Dijkshoorn and Visser, 2011]

2.1.2 Quadcopter

As we established previously, a quadcopter is one model of a UAV. Its lift is produced by propellers, arranged in a cross fashion. It is impressive that this cross configuration was initially proposed in 1907, as a prototype, and it has held strong since then [Digital Trends, 2018]. Such a simple design, yet so effective and reliable. Presented in this section are all specific characteristics for this type of system, the modern quadcopter.

Figure 7: Example of an octocopter, used to take stable photographs. [Eschmann et al., 2013]

First of all, a quadcopter is by extension a vehicle. This characteristic implies it must carry some sort of payload during flight. This includes its own components, fuel, structure, and any other added cargo. In some applications, a drone may only lift itself for best maneuverability and energy efficiency. However, in other contexts the primary function of the quadcopter is to transport equipment or goods. The application of the device will most of the times decide the kinds of payloads the drone must carry.

It is important to consider the size, positioning, and weight of said payload to best decide over control and limitations of the vehicle. Too heavy, and it will require huge motors or batteries to lift off. Too big, and it may have trouble avoiding collisions. Poorly positioned, and the drone may drift or tilt too much to one direction. The payload is a constraint to treat carefully for each build, because it will set the general necessities for the hardware.

Secondly, the quadcopter is composed of several construction pieces or hardware. This is a list which is reduced to the basic elements but is not limited to the mentioned components. This list is based on the type of drone used for this work, which is, as mentioned, a modern quadcopter.

Structure The base component for all drones is its structure. It is where all the other hardware is mounted on. For quadcopters, it usually comprises of a small box or plate at the center, and four bars or pipes extending from the center on the same plane in a cross fashion. At the center is where most electronics are held in place, and the motors usually rest at the far ends of the bars. The material from which

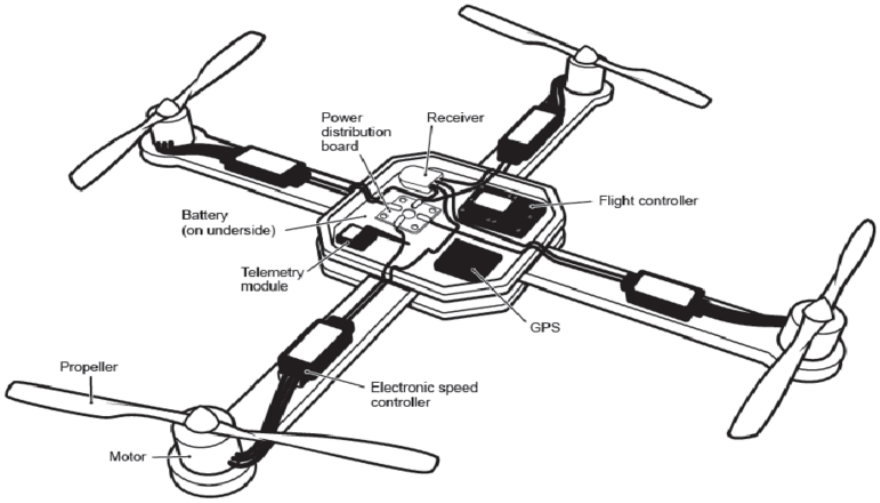

Figure 8: An example of a quadcopter as designed. [Pothuganti et al., 2017]

a structure is made, needs to be as light as possible. This is to reduce the weight of the UAV and waste less lift on counteracting the force of gravity. Common materials for structures are plastics, carbon fiber, glass fiber, and even cardboard of wood.

Propeller As mentioned before, a quadcopter must have a set of four propellers. These are generally separated similar to the vertices of a square and on the top plane of the vehicle. Although there are inverted (bottom) configurations, still a top one approach is the most common one. Each propeller has a profile, which is the inclination (or pitch) of the main lift surface. The angle is measured relative to the horizontal plane, and the higher is the angle of attack, higher the lift produced, but also higher the power demanded.

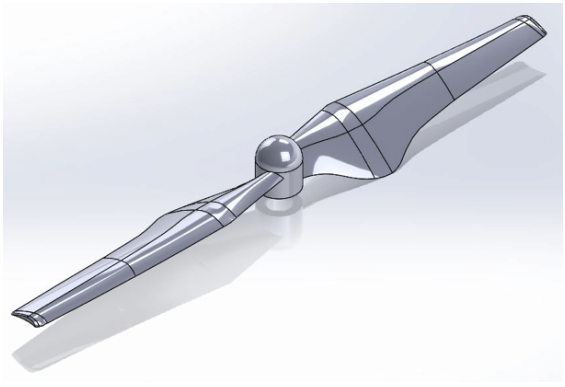

Figure 9: Example of a drone propeller designed in CAD. [Belikovetsky et al., 2016]

Motor Most quadcopters have four motors fixed to their structure, one for each propeller. This enables the aircraft to have individual control over the velocity of each propeller, and achieve different kinds of movement when varying the velocity. Although most quadcopters have the propeller directly mounted on the motor's shaft, some designs mount it on a separate axis and transmit the motor's power via cogs. High speed motors are the most used due to propellers' low profile. This grants higher stability in exchange for an slower reaction time.

Battery The most common source of power for the modern quadcopter is a high-energy-density quick-discharge deep-cycle battery [Edward F. Hogge and Quach, 2015]. The front reason for this is because energy density refers to how much energy can be stored in relation to the space available. So, if we have a higher energy density, we can store more energy in a smaller package, and thus, fit a bigger battery. Drones demand several Amperes of current during normal flight, so energy consumption becomes a concern. Bigger batteries have to be dimensioned for longer flights.

In addition, power demand from the motors is quite large, ranging from a few Amperes when still, up to dozens of Amperes when maneuvering. So, it is no surprise that a quick-discharge battery is needed. Normal batteries are not able to withstand high current discharge profiles, nor designed for surges or spikes of power. So, to equip a battery with decent resiliency is a must.

A deep-cycle battery has the capability of depleting its energy storage almost completely. In principle, batteries must retain some charge to protect themselves from complete exhaustion, and must be cutoff to avoid damaging them. Most commercial batteries have a cutoff at a relatively high percentage, and cannot take advantage of all energy stored in them. In contrast, a deep-cycle battery is able to almost exhaust itself completely before it needs to be cutoff. This increases the amount usable energy, and reduces the amount of batteries required.

Controller All quadcopters are equipped with a control card or a flight controller. It is usually connected to the battery, or a regulator. It is the one in charge of taking all logical decisions (process information). To paraphrase, its main function is to maintain control of the aircraft through the calculation of the necessary changes, and then translating them into electric signals for the rest of the vehicle. The signals produced are: throttling the motors, reading and writing information, managing communication, to mention a few. All other electrical subsystems in the drone are typically connected to this "brain". Most programming and control algorithms are implemented in this controller, with some exceptions.

Sensor Sensors are the electronic components which permit the device to receive information from the world. Measures made by these components are passed on to the controller. Common sensors found in a drone are: ultrasonic, magnetometer, accelerometer, temperature, GPS, altimeter, camera, and voltmeter. Based on

these measures, the flight controller can take decisions and maintain the drone in flight.

Radio Modern UAVs are fitted with an electronic radio system to communicate. Control, image, and even sound information is sent and received through a radio, making it the only way a base station can undestand what the drone perceives. Many technologies are used, like WiFi, Bluetooth, 3G, LTE, and FPV systems. It is not compulsory to have a radio system, but for modern applications, like filming and racing, it is a must.

Power We talked about the logic computer in the drone, which is in charge of controlling the drone. However, there are other components who complement the brain. Let's call them the muscles of the drone. An *electronic speed controller* (ESC) is the electronic piece which takes the brain's decisions on the motors and controls how much power the motor actually receives. There is an ESC for each motor, and is directly connected to the battery to draw power.

This list covers the basic components that constitute the modern quadcopter. We are assuming the vehicle covers basic drone functionality such as translating, preserving stability, and sending/receiving data. Most cases found as commercial products have these components in one form or another, but besides, other designs may also exist which implement other functionalities.

2.1.3 Flight Principle

Now that we have covered its composition, we can talk about how the quadcopter flies. In this section we describe how it can take off, as well as how it achieves movement, and by extension, control.

To start off, let us view vertical movement only. For that, we need to start with the Third Law of Newton. "For every action (force) in nature there is an equal and opposite reaction." [Hall, 2015b]. Taking this statement to drone mechanics, a drone to lifts off, when upwards acceleration is more than $9.81 m/s^2$ $[g]$ or its total net acceleration $[a]$ is positive. As Newton states, for a body to move in one direction, an equal force $[f_{Drone}]$ must be exerted in the opposite direction. Returning to case, the quadcopter must force something down, and the resulting force must be enough to counteract gravitational force.

$$a_{Drone} = \frac{f_{Drone}}{m_{Drone}} - g$$

As mentioned before, it is important to consider the drone's mass $[m_{Drone}]$. Since the acceleration of the drone is inversely related to its mass, a heavier drone will limit its flight capabilities. Also, as stipulated earlier a quadcopter generates lift (perpendicular upwards force) through the rotation of its propellers. The faster a propeller spins $[\omega]$, the more lift $[f_{propeller}]$ is generated [Hall, 2015a].

$$f_{propeller} = K_{propeller} \times \rho \times A_{effect} \times \frac{\omega^2}{2}$$

$$f_{propeller}(\omega) = K_{lift} \times \omega^2$$

We must remember that propellers can have different designs, which will determine its specific lift coefficient $[K_{propeller}]$. Also, air density $[\rho]$ and propeller area of effect $[A_{effect}]$ influence this relation. But, if we simplify all constants in the model, we obtain that the angular velocity of the propeller is the main factor influencing lift. The resulting equation for vertical acceleration is the following.

$$a_{Drone} = \frac{4K_{lift}\omega^2}{m_{Drone}} - g$$

Note the multiplier by 4, which is for each of the 4 propellers in the quadcopter.

If we were to hover the drone in place, the net acceleration is 0. Although it is true for vertical movement, it is not completely true for z rotation. The complete story is somewhat more elaborated. It must be assumed that a propeller is non-ideal, so it produces a torque $[\tau]$ on the drone in the same direction it rotates as a byproduct. If we set all propellers to spin in the same direction, the resulting torque would be a net positive or negative value. If we assume all propellers rotate at the same speed, that scenario would be:

$$\tau_{drone} = 4\tau_{propeller}$$

This would send the drone rotating uncontrollably, since there is a rotational force different from zero. This is the reason why quadcopters always operate with an even amount of propellers, and half of them rotate in one direction while the rest in the other direction. That way, a rotation control is achieved, and the drone stops rotating out of control. Returning to the same scenario:

$$0 = 2\tau_{propeller} - 2\tau_{propeller}$$

Rotation stability is achieved, as long as these two conditions are maintained. This state of hovering in place is the drone's most stable state, since it is capable of maintaining this position for extended amounts of time. But, what if we want to move the drone? We know we can move up and down just by changing propeller rotation uniformly. Upwards acceleration exists when net acceleration is positive, and downwards acceleration when negative.

Nevertheless, if we start accelerating propellers with non-uniform patterns, interesting new behaviors emerge. We can observe movement and rotation in all directions, which also shows the superiority of this type of UAV over others. As defined in aviation

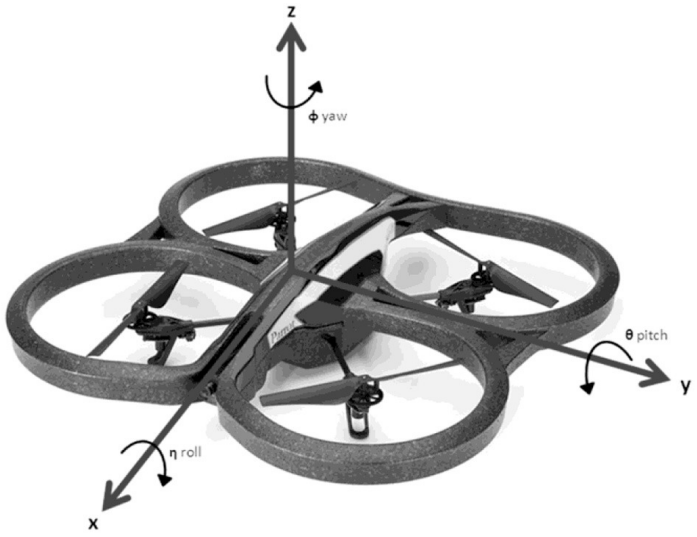

Figure 10: Axis convention for X configuration quadcopters. [Darío Maravall and Fuentes, 2017]

jargon, a drone is capable of rotating along its three axis, meaning it can modify roll, pitch and yaw; and also translating along the three axis X, Y and Z respectively. These six possibilities is what comprises the drone's six degrees of freedom, so it is technically capable of achieving every possible position.

Tilting (to pitch or roll) is achieved when two propellers in the same side accelerate more $[\omega_+]$, and the two others decelerate in the same proportion $[\omega_-]$. From a net acceleration point of view, total lift is not modified. But, when we analyze from a torque perspective, an unequal torque in X or Y axis results as the propellers lift non proportionally.

Z axis

$$0 = 2f_{propeller}(\omega_+) + 2f_{propeller}(\omega_-) - g$$

Z axis

$$0 = \tau_z(\omega_+) - \tau_z(\omega_+) + \tau_z(\omega_-) - \tau_z(\omega_-)$$

X or Y axis

$$\tau_{x_{drone}} = 2\tau_x(\omega_+) - 2\tau_x(\omega_-)$$

To rotate in the Z axis (yaw), we rely on the motors' non-ideal properties. Parting from the hovering effect, the torque can be modified in the same manner as in the tilting process. We increase speed in two same-turning-direction propellers, and reduce speed in the other two. This creates an unequal balance in torque, enabling angular acceleration.

Z axis

$$0 = 2f_{propeller}(\omega_+) + 2f_{propeller}(\omega_-) - g$$

Z axis

$$\tau_{z_{drone}} = 2\tau_z(\omega_+) - 2\tau_z(\omega_-)$$

X and Y axis

$$0 = \tau_x(\omega_+) + \tau_x(\omega_-) - \tau_x(\omega_+) - \tau_x(\omega_-)$$

Translation along X and Y is the most complex, since it requires the drone to be already at a tilted angle. While in hovering position, it is only counteracting the gravitational pull. However, when tilted at a certain angle, gravity is counteracted and a residual force is applied in the perpendicular direction. This way, the only resulting acceleration is in the perpendicular plane to the Z axis. For example, when pitched forward and stabilized, the quadcopter rests at an angled Y $[\theta]$ inclination. Accelerated propellers stabilize the system's rotation and Z position. As a consequence, it moves forward, due to the residual lift in the forwards direction.

Z axis

$$0 = 4f_{propeller}(\omega_+)\cos(\theta) - g$$

X axis

$$f_{x_{Drone}} = 4f_{propeller}(\omega_+)\sin(\theta)$$

Z axis

$$0 = 2\tau_z(\omega_+) - 2\tau_z(\omega_+)$$

X and Y axis

$$0 = 2\tau_x(\omega_+) - 2\tau_x(\omega_+)$$

These are all the possible ways in which a drone can move. The combination of all is the key for more complex acceleration patterns. Nonetheless, acceleration initiates movement, but changes can only be seen when time passes by. Moreover, velocity $[v]$ and position $[p]$ are the result of accelerating during certain amount of time. They must be differentiated, since they are strictly related and often confused. Velocity is a dependent variable on time and acceleration, and position is the absolute reference in which a measure can stand, which is directly influenced by velocity and time. Here is a simple dimensionless set of relations:

$$v_{i+1} = \int_{T_i}^{T_{i+1}} a dt + v_i \rightarrow a(T_{i+1} - T_i) + v_i$$

$$p_{i+1} = \int_{T_i}^{T_{i+1}} (\int_{T_i}^{T_{i+1}} a t dt) dt + \int_{T_i}^{T_{i+1}} v dt + p_i \rightarrow \frac{a(T_{i+1} - T_i)^2}{2} + v(T_{i+1} - T_i) + p_i$$

To summarize, this is how a quadcopter can lift off and move. It takes off from the lift principle, and applies it to its spinning propeller blades. Depending on how fast these propellers go, is how lift force varies. The interaction between its propellers and the environment is how the quadcopter can achieve movement. Finally, position and velocity are tightly related to acceleration, and must be assumed as dependent among them.

2.1.4 Normativity

As we know, mainstream usage of drones spiked a few years ago. A wave of new small unmanned aircraft, understood as public property, started occupying the skies. This opened the door for a whole world of possibilities, including awesome demonstrations of skill and engineering, but also a whole lot of new dangers. That was the phenomena which started the whole revolution in both aviation legislation and safety requirements.

UAV legislation has been treated differently across the globe. Countries with very proactive military forces were the first to realize the perils behind public UAV usage. Anyone could put any payload in the sky, fly it kilometers or miles away, and possibly deliver it without ever being on site. Also, these aircrafts could get to reserved airspace and cause havoc while irrupting normal air and space operations. The consequences of any of these could be catastrophic. So, those countries started a new defense strategy focused on limiting use of these devices to safe usage only. That is how finally in 2014 regulation got enforced.

However, in our home country, Mexico, legislation was established three years later from the rest, mainly as a result from foreign law adoption. The official law was published in Autumn 2017 and honestly, is quite detailed. Registration for all drones is obligatory. Drones were split into some mixed usages: private recreation and private commercial/non-commercial. Recreation is understood as general amusement and entertainment exclusively, while commercial/non-commercial use is for any other application including photography. We must note the law does not include State property, neither completely robotic UAVs.

Some weight restrictions were implemented:

- **Micro** with a weight of less than 2Kg. Max operation height of 122m. Max operation distance of 457m.
- **Mini** with a weight greater or equal to 2Kg and less than 25Kg. Max operation height of 122m. Max operation distance of 457m.
- **Macro** with a weight greater than 25Kg.

The best explanation for weight restriction is understanding the fact that a heavier drone has bigger and better components. So, as weight increases, so does the hauling capabilities, propeller size, and possible functionality of the device. Better piloting and regulation is a must to drive bigger drones, leading to a need for more certified and responsible pilots.

Furthermore, other restrictions apply:

- Pilots and drones must be at least 5 nautical miles away from any airport.
- Pilots and drones must be at least 0.5 nautical miles away from any heliport.
- No item should be allowed to fall from an aircraft even if it is protected or has a parachute.

- Responsible flight is a must, as well as preliminary aircraft inspections before takeoff. Sensible condition testing is encouraged.
- Control of the aircraft must be present at all times. No uncontrolled drones may be in flight.
- Manned aircrafts have priority over UAVs.
- Sunlit flights are unrestricted. For nocturnal operations, special authorization is required.
- Only national pilots may operate an UAV.
- Only one UAV may be operated at once.
- All drones have recreational max operation height of 122m, and a max operation horizontal distance of 457m.
- Cloud avoidance is a must, maintaining visibility of the vehicle at all times, at least 1.5Km in all directions.
- The *Autoridad Aeronáutica* is the authority for these regulations and the one in charge of enforcing them. Autoridad Aeronáutica is understood as *La Secretaría de Comunicaciones y Transportes*, through *La Dirección General de Aeronáutica Civil*.

Turning on to this year, 2020, laws have not changed as much, according to current norm NOM-107-SCT3-2019 [COMUNICACIONES Y TRANSPORTES, 2019].

2.2 Simulator

Most of this work was done in a simulation environment. This section will provide the base concepts to comprehend how this technology works. Also, it presents a quick introduction of its functionality and capabilities.

2.2.1 ROS

The Robot Operating System, commonly referred as ROS, is a open source platform, completely oriented to robotic software development. As stated in its wiki article, it is intended for simplifying the creation of a robotic system. This is the base platform on which our simulation is built.

> "ROS (Robot Operating System) provides libraries and tools to help software developers create robot applications. It provides hardware abstraction, device drivers, libraries, visualizers, message-passing, package management, and more. ROS is licensed under an open source, BSD license." [ROS.org, 2018]

From this paragraph, it is understood that ROS's aim is to simplify the base building blocks for higher robotic software development. In other words, it removes the hassle of creating all code from zero, or the load of developing new drivers for each hardware piece. Instead, one can conveniently download plug-and-play libraries and integrate your devices to the ROS environment easily. Therefore, faster and more abstract implementations are possible, and the focus can shift towards developing technology instead of integrating at low level.

ROS is a platform which runs on a computer. It is an operative system by itself, because it manages hardware and exposes it for usage of application. However, typical implementations are based on other operative systems (OS) like Windows or a Linux distribution. This means that it is managed by another OS, but it carries its own tasks and can start processes for other tasks. We can consider it like a second level OS, or maybe an app-OS.

Then, why it is different from a common OS like Linux, since Linux is also an open source OS with splendid hardware support? Well, the answer is that ROS intends to work on a much smaller scope, since it is specialized for robotic development. But most importantly, its paradigm is completely different from an standard OS.

ROS works internally like a network. There are *nodes*, which function like stations. Each node is assumed as an individual entity inside ROS. Usually, an application or a computer driver has a node running in ROS. This enables the use of its resources by the rest of the ROS environment. However, just like a network, messages need to be sent in order to exchange information between nodes. These messages are first published to specific *topics*. Then, nodes can subscribe (or listen) to these topics in order to obtain the updates. This way, when a message is published to a topic, all subscribed nodes receive a copy of it for processing.

Moving on, there is always a central node in a computer. This node is called the ROS master, which is the one of coordinating all ROS interactions in the computer. It is the main process that makes ROS posible. This node has a few different tasks:

- Registering the new nodes created by other processes.
- Keeping track of all topics (creation, change and deletion).
- Communicating with other masters.
- Servicing nodes.
- Receiving, replicating and delivering messages.

It was mentioned there can be only one master per computer, but still there is also the possibility of aggregating several computers. We can achieve this through the Internet or just a local network. This way, masters can mesh together to form bigger ROS environment.

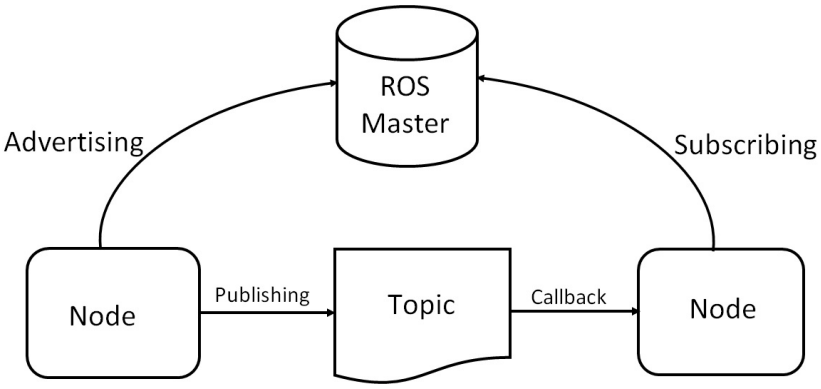

Figure 11: Typical operation of a ROS environment. [Wilcher, 2019]

Diving into details, there are some special rules we have to consider. Topics have a defined message type, and messages sent to that topic must respect that message definition. Additionally, messages can have any payload, but must preserve its predefined structure and data types.

As a creative exercise, we can also imagine ROS's network similar to a real Ethernet network. There are hosts within the network (nodes), which transmit standard Ethernet frames (messages). A central switch (master) is in charge of maintaining the VLAN table (topics) and receive the frames. Whenever a new frame is received, the switch replicates and sends it only to the corresponding VLAN hosts. Switches can be aggregated in a similar fashion ROS masters can be aggregated.

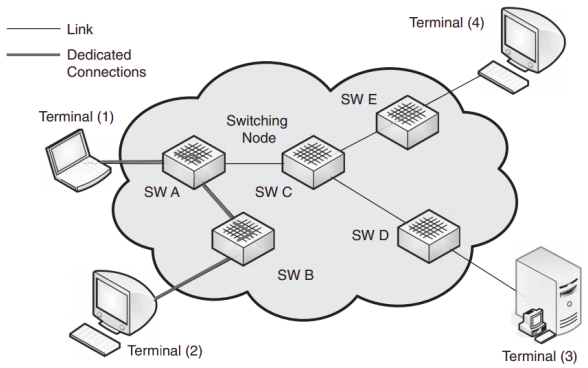

Figure 12: A circuit switched network is similar to a ROS environment. [Farahmand, 2020]

2.2.2 Gazebo

The actual simulation of physics is done in a separate software. It is named Gazebo, also from an open source initiative from 2004 up until 2011. It is a proper physics simulator, which is focused on robot simulations. Its popularity started with the rise of

modern robotics and is still used widely in robotic research. It is still in development and support to this day.

Figure 13: Example of a Gazebo world. [Furrer et al., 2016]

It is comprised of four main parts: the world, the models, the server and the interface. [Open Source Robotics Foundation, 2014] Each part plays an specific role for the simulation. A model is any one of the simulated "things", just like a drone or a wall. The world is the description of all the rules that the simulated environment will follow, and also includes the models and the general description of the terrain. The server handles the information from the world and calculates all the simulation in real time. Finally, the interface draws the video of the resulting calculations and displays it on a window.

Gazebo has the capability of integrating with ROS, which means it can simulate hardware for ROS consumption. Devices in the simulation can be accessed via the gazebo nodes, which the simulator creates and listens to in ROS. On top of that, data is generated inside the simulation and picked up by virtual sensors. Just as real sensors, it measures the world and passes on the data to the ROS node. This integration lets the simulated device be controlled and supervised from the ROS platform.

2.3 Control Theory

Control refers to the ability of modifying the behavior of a system. Control theory is a vast field of study, and is typically applied to engineering in the form of autonomous control, or feedback control. Its purpose is to drive a system to meet a given reference without the interference of another system [Franklin, 2002]. That is to say it simple, but in reality is more complex.

A target system or controllable system is defined as the closed object of study, or if you wish to call it, the interesting device you control. We can conceive it as a "processing box", meaning that something goes into the box, and something comes out

of it as a result. It is not always important to know exactly how the box works, but it is critical to know what it takes as input and what it throws as output. The inputs are all variables we can alter to have an effect on the system, while the outputs are all the variables that the system can modify. An input influences directly the output of the system, and is continuous as long as the controllable system does not change.

A car moving forward would be an example of this. We can consider it a system because it has a set of boundaries in which we can define it. In other words, we include only the car and maybe the road it is on. On top of that, the car has a gas pedal and a brake pedal to command movement, which we can consider the inputs. The variables which the car modifies are the distance traveled, the current speed it travels at, the acceleration felt inside, among many other variables. A system may have a huge amount variables as output, but for a control standpoint, only a handful may be observable or interesting [Franklin, 2002]. If we make the exercise of starting the car, and pressing the gas pedal, we will observe all outputs change. The position of the car shifts as it moves; speed increases; we would probably feel come acceleration into our seats; and so on. This stays true for all controllable systems.

Moving on, a complete control system includes the target system, and is completely devoted to condition said system. The control system sets the inputs, and at the same time, the outputs of the target system are viewed as well. Sometimes, *PID* (Proportional, Integrative, Derivative) control is added to the system. A PID control is a mathematical tool used to modify the inputs before they enter the controlled system, with the intention of influencing the output characteristics. It does not change the controlled system completely, but it does help achieving certain criteria in the output. Some of the output characteristics it can modify are: time in which the desired output is matched, overshoot and undershoot, and response aggressiveness.

We previously discussed that a control system could be one of two kinds: open loop or closed loop. The *loop* term is only to reference how it looks in a diagram, and it refers to the ability to influence itself. Anyhow, it was mentioned that an open loop does not take into consideration any of its outputs to influence its inputs, and that it essentially is not useful for robotics since it ignores the outcome. Furthermore, a closed loop is the one that does change its inputs based on at least one of its outputs, and that characteristic is what makes it so useful. We gain the ability of setting desired outputs and letting the system try to reach those outputs instead of doing that ourselves.

A closed loop control system has more elements to it than an open loop. Feedback is done through a translation or *transducer* element. This means that the outputs are transformed into input readable measurements. Then, an error is calculated, which is only the difference between the actual input or reference of the system and the transduced (or estimated) one. In simple terms, the error just compares what the target system feels it got and what we actually introduce to it. Finally, we take that error and reintroduce it to the controlled system (or PID if it has one). The control system constantly shifts its input until the error gets reduced to zero. This way, we guarantee the real output is the one we desire.

If we take the car example and convert it to a closed loop system, we must add some

more elements. Let us say, we set the speed of the car. The car must have sensors on the wheels which digitally tell the car the actual speed. Then, we must know how fast we want to go and set it on the car computer. The rest is handled by the computer. The system input is the set speed, and the error is the difference between the real and the set speed. What the computer will do is to introduce the error as the virtual "pedal press", so that the car eventually reaches the speed. This way, once the speed is matched, we will continue to travel at that speed.

Finally, we can visualize control systems through several tools [Franklin, 2002]. Classical control (mainly used until 1980s) uses frequency (Laplacian) based models to study control. It transforms the complete system into the mathematical frequency domain, and operates from there. Outputs are directly matched to inputs in a single equation which comprises the entire system. This approach is is very good to study the response of said system, but lacks visibility into the internal status of the variables. In contrast, modern control (widely used today) studies systems from a state space perspective. All variables are calculated in terms of its next state (step in time). This makes it more complex to analyze since it has equations for every observable output, but it also has the advantage of complete variable visibility.

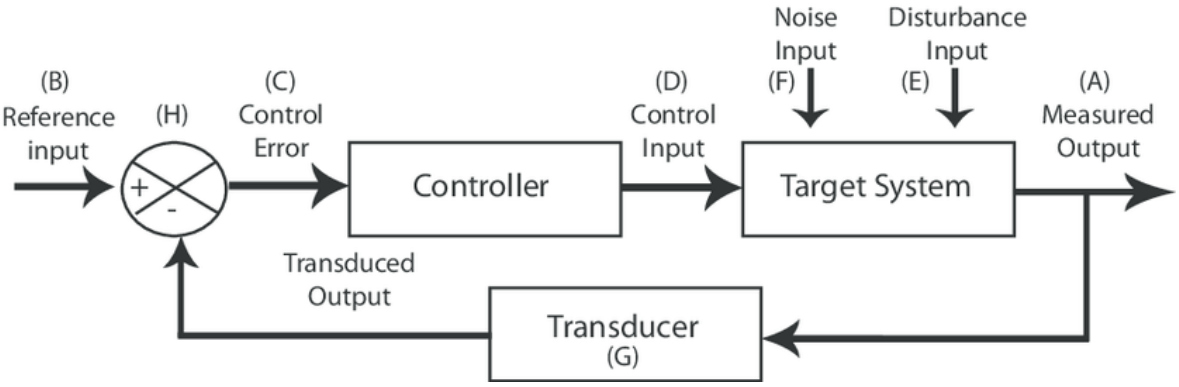

Figure 14: General configuration of a control system diagram. [Villegas et al., 2012]

To have a better understanding of the system, we make diagrams like the one showed in the last figure. It enables us to have a better idea of how the system actuates on itself, and see more easily how sequencing takes place. If we take a deep look into it, we can observe all previous elements are present. Just note "(H)" is the calculation of the error and "Controller" could be the PID controller we previously mentioned.

2.3.1 Fault Detection

Control is all about achieving outputs and modifying inputs, right? The system is given, and we set up our controllers to match our wanted references. Everything operates normally. Ideally, all systems are have no modifications through time and operate in a known range of outputs given our normal inputs. But, what happens when we work with real life scenarios? Systems are prone to fail sometime. It is not necessarily because

the system itself changes, but because operating conditions are no longer the same. It usually happens far when the system first debuts, and has endured many hours of work. But even for that, engineers must foresee these scenarios and plan ahead.

To define a fault, we take a look at the outputs of the target system. They are always expected to oscillate inside a known range given a known input. For example, a car, it is always expected to reduce its speed when we are completely off the pedal, and eventually come to a stop. However, when that does not happen, we know something is wrong (and may panic a little). The same goes for other scenarios. A system is considered to be faulty when an output is out of range and does not return to normal operation range after a certain amount of time. It does not matter if the system returns afterwards to normal operation. If the system maintains a faulty range for an extended amount of time, we can consider that a fault, even if it later disappears.

Faults are serious. They are the most dangerous operational states for systems, since most systems are not designed to work in those extreme conditions. If any faulty operation is prolonged enough, it will most certainly end up damaging the whole system. They have a limit, and we must try to detect faults in time to avoid this damage.

There are several reasons why a fault may appear in a system. The difference resides in where the anomaly appears inside the system. There are several parts where it can be found: the actuators or control inputs fail to meet the intended input; the model used to mimic the system is not capable to match exactly the behavior of the real system; the parameters used for the model are not calibrated to the act adequately on the system; the sensors are picking up measurements which are not accurate; the control inputs are affected by an additional input... These are only a few to mention, since a system is a complex and closely related unified piece. The key to identifying faults is to make sure we measure as many parts of the system as we can. The more information we have, the better we can detect faults.

There is a very well-known YouTube video in which a CD (Compact Disc) is spun at ridiculous speeds [The Slow Mo Guys, 2015]. The normal operation speed for a CD is at maximum 500 RPM. However, in this video, the CD rotates at a much higher speed. It holds well for a few seconds, but after that, it shatters completely. It was not designed for that extreme, even if it normally rotates at high speeds. This same principle applies to all other systems.

So, to detect faults, we must supervise or monitor the inputs and the outputs at all times. There is always an expected output for a certain input during normal operation. So, whenever the system is detected to fall out of the expected range for an extended amount of time, a fault alarm must be triggered. To know which should be the corresponding output for a given input, we have an abstract mathematical model that mimics the one in the system.

For this project, we use the threshold method to detect. The process is simple, we use thresholds as the limits of the normal operation. If it stays out of range for too long, then we trigger an alarm. This is an example of an algorithm to detect it: first we set a threshold for both measurement deviation and time deviation. Then, we compare once the observed output to the modeled expected output. If the difference between both

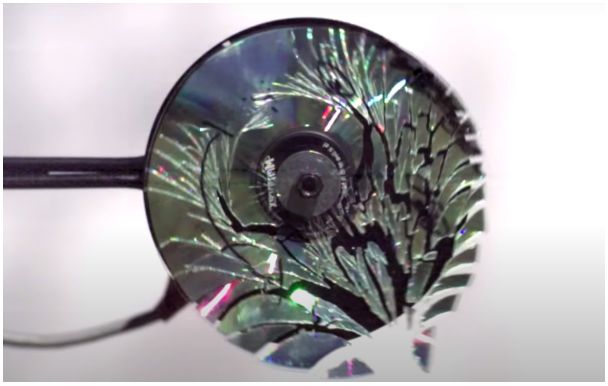

Figure 15: Breaking CD due to excess rotation speed. [The Slow Mo Guys, 2015]

measurements is greater than the measurement deviation threshold, we start counting time. If the measurements stay greater that the threshold, we keep adding more time. The moment the cumulative time surpasses the time deviation threshold, we can trigger a fault condition or alarm. If at some point, the threshold is not surpassed, we reset the time or subtract from the counter. Otherwise, we say that there is no fault. The whole point of the method is that this way we can check whether the model and the system are coherent [Verde, 2013], or otherwise we trigger the alarm and inform about a fault.

2.3.2 Fault Tolerant Control

Fault tolerant control (FTC) is the result of integrating fault detection and correction mechanisms to a control system. The aim of this implementation is to maintain system operation even after a fault occurs. It is expected to see reduced performance after fault correction scenarios, but the system will try to maintain operation as best effort. It is a completely different approach to fail-safe systems, which shutdown as soon as a fault is detected, or fail-operational systems, which can operate even if a fault is present [Blanke et al., 2000].

Fault tolerant control needs fault detection in order to operate. As soon as a fault is detected, a new algorithm triggers and the control itself is changed. There are many control types which are FTC, ranging from robust control to adaptive control. Each one has many subcategories, and all of them have a rich expertise background we could explore. However, to keep this document brief, only one is presented.

The selected approach for FTC is feedback adaptive direct control. Feedback refers to the generation of useful information from the outputs and consumed by the controllers to create a new response. Adaptive refers to te capability to use the same system model for estimation, so it may not assume at anytime the controlled system changed in any way. Direct refers to the use of corrections directly on the controller, but not to calculate parameters. This is the method we will use as reference for the rest of this work.

3 Project Description

The project here presented is a take at implementing fault tolerant control (FTC) in a quadcopter. The whole implementation of this control is though an algorithm contained in ROS. Moreover, a Parrot AR.Drone UAV is simulated inside Gazebo, as a testing laboratory for the control algorithm.

The simulated hardware (drone and simulated space) is visible from the ROS environment; this is with the purpose of both sending commands to the drone and reading from Gazebo's sensors. The rest is performed by the programs running in ROS. These are in charge of computing the calculations for controlling the drone and other extra tasks.

The whole project can be thought of as a hybrid system. On one side, we have the basic stability control of the drone, which comes by default. And on the other side, we have a whole new complementary control supplied to the drone from the outside, similar to a redundant control loop. It may seem difficult to grasp at first, but it will become clear in the following paragraphs.

3.1 Elements

There are several elements which compose the project. Both physical hardware and ROS nodes are included in this list, due to their nature of being separate entities for ROS. This section describes them generally, including their general characteristics and functionality. Nonetheless, it is not explained how they work together in the control process. For further information on interactions, refer to the next section "Control Models".

Computer The first piece of real hardware used is a computer. This is where all the simulation of virtual hardware is done, and the containment of the local ROS environment. Also, all peripherals like the display and controllers are attached to this system. The OS version installed for this project is Ubuntu 16.04 LTS (Xenial). Device model is not relevant, because of Linux's immense versatility and capability to run almost in any 64 bit machine.

Controller The controller is the other piece of real hardware attached to the system. This is capable of sending the human machine inputs for mobility and is a convenient way of maintaining control of the algorithm in case something goes wrong. For documentation purposes, a gaming Xbox One controller was used.

ROS The Robot Operating System was introduced in an earlier section. This is the platform in which all the implementation of the control system is done. The version that best matches the OS election is ROS Kinetic Kame. It was specifically released for Ubuntu Xenial and is currently supported. Its end-life is expected to be the same as the OS. Documentation can be found at the official page [ROS.org, 2016].

joystick_drivers This joystick library is the first one used inside ROS. It enables driver or register inputs detected in Ubuntu to be readable by the ROS environment. It runs a single node called *joy_node*, and publishes to its */joy/** topics. Controller information from the Xbox hardware is easily readable from here. There are many implementations for controllers in ROS that use this driver. Documentation is obtained in its ROS documentation page [ROS.org, 2011].

Gazebo Also introduced earlier, Gazebo 7 is the simulator used to run the virtual AR.Drone hardware and its environment. We can access the information from the whole simulation environment just by subscribing to the topics available in ROS. Gazebo publishes to the */gazebo/** topics and has several nodes running at a time.

AR.Drone-ROS This library works in coordination with Gazebo, but from the AR.Drone perspective. Gazebo does not come with a drone model ready for flight. This is why this library is crucial. The library adds the model simulation to Gazebo and extends its visibility to ROS. That is why the simulation is capable of replicating the drone's behavior, as well as providing sensor information that would come from it. The drone's node, however, is the same as the simulator. Gazebo reads from the */cmd_vel* topic and feeds those commands to the drone. Meanwhile, the information generated from the drone is fed to a bundle of different topics, including Gazebo's original topics. Documentation for this library is found in its GitHub page [eborghi10, 2016]. Please note that this is an updated version of the original tum_simulator library, which its documentation and credits are found on its ROS documentation page [Huang and Sturm, 2018]. Other dependencies of this library are related to the original ardrone_autonomy library, which is used to communicate with a real AR.Drone [Monajjemi, 2015].

The inputs accepted by this specific library are the same as the real drone: all movement is given in desired velocity directions. You may only provide target speeds to the drone in any of its available axis. This means that acceleration and position is not possible to pass on to the drone by standard means. So, for example, if you intended to move it forward, a positive linear X axis speed would need to be provided. Respectively, to turn left, a positive angular speed is set. The only axis we use in this project are linear X, Y and Z, and angular Z because all movements and stable positions are achievable with these commands. Angular X and Y are not used because they are harder to predict and do not leave the drone in a stable position if altered by themselves.

ardrone_control This is the main library created for this project. It includes all control algorithms and processes the system uses to maintain and recover the drone, even though it is a single node implementation. Its node is named *ardrone_control* inside the ROS environment, and reads from and writes to several topics. The code was written in C++, and has a functional oriented approach. It is the machinery that drives the fault tolerant control for the drone.

3.2 Control Models

The project was developed in stages, integrating new functionality per new stage. At first, it started with a simple open loop model with the purpose to test node functionality. In principle, this could be taken as a simple controller implementation, similar to a human doing manual control. However, as the project grew, more control algorithms and concepts were integrated making it more robust. The end result was a fault-tolerant system.

In this section, all stages are explained in depth. At the end of each stage, there is a figure with an equivalent diagram for that implementation.

Open loop The first model implemented was an open control loop model. It can be thought as just a remote control. A drone is moved by a human, similar to an RC car. Inputs go through the control and end up impacting the drone respectively. Responsibility for referencing flight and paths is solely left to the driver.

It seems easy enough. Nevertheless, the implementation of such in ROS requires a few extra steps. First, the inputs are captured by the computer's hardware, and subsequently passed to ROS by the joystick library. The published topic is then read by our ardrone_control library and mapped to a series of actions depending on the input. The four axis movements or control inputs (4 U) are calculated, and then then fed to the command topic. Afterwards, Gazebo proceeds to move the drone and gets displayed by its interface. The user, as driver is in charge of supervising the drone's movement at all times while using this control model.

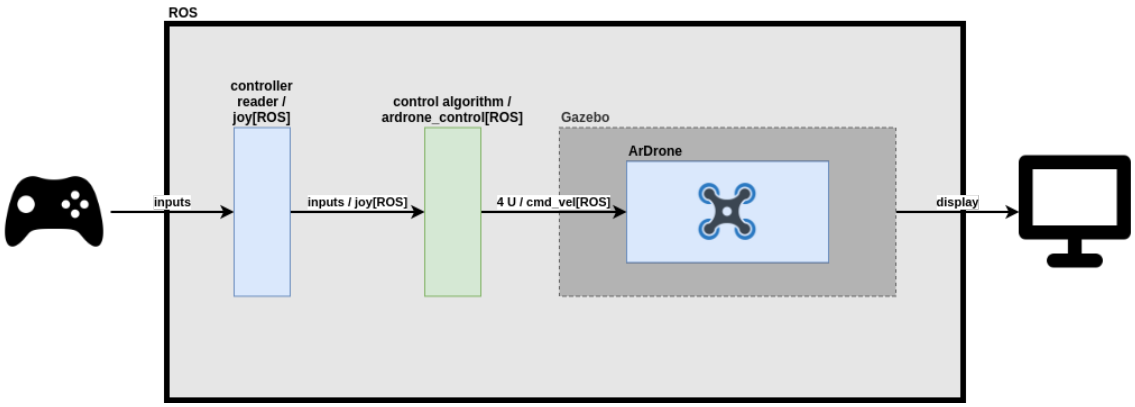

Figure 16: Open loop model for AR.Drone control in ROS.

Input calibration Open loop control is good for a toy, but it does not have any way to mitigate defects. In real world applications, a drone is prone to have a little deviation in its open loop control. a small sway could be considered as a small fault, maybe because of wear or part tolerance. If the sway is perceivable, the user can manually say the fault is present and try to compensate for it. However, there is no

active state of compensation that way, and as soon as the user lets go of the controller, the sway returns.

What this control model offers is a way to compensate manually for this small fault. There is no FTC integrated yet, but it is the first attempt at implementing some kind of adaptive system, even if it is manual. What a user can do is tell the control that compensation is to take place. Then the ardrone_control node starts to read a separate set of inputs from the user. Whenever the user feels comfortable with the state of calibration, it can tell the ardrone_control node to save that. The end result is that calibration takes place and the control algorithm now compensates for the sway present.

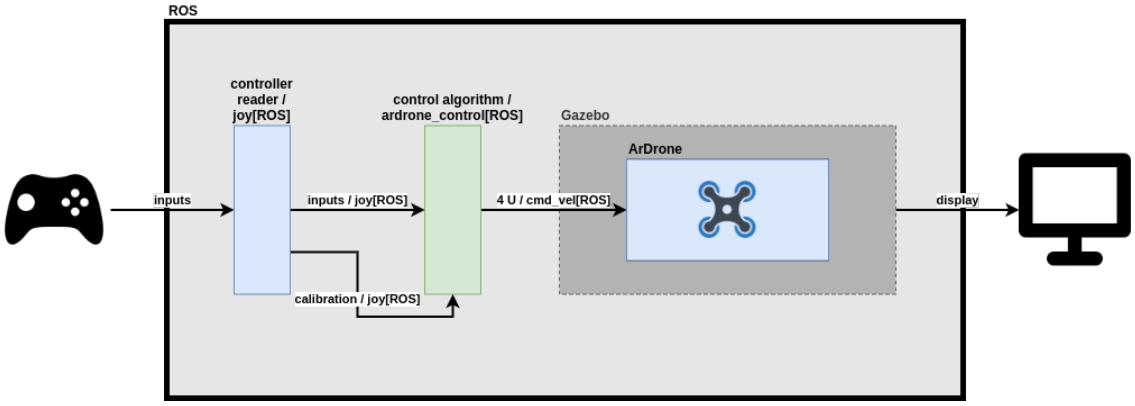

Figure 17: Open loop model with manual constant compensation.

State estimation This stage is the start of closing the loop. A state estimator is the new module present in this stage. What it does is it first reads the absolute position provided through Gazebo's sensors, published under the /gazebo/model_states topic. This is similar to how a camera or tracking system would work, but virtually. Then, using simple derivative methods, the estimator obtains estimated velocity and acceleration and makes all three available for future use.

The estimator can be considered the transducer or "observer" of the system, since the data produced is later read by other modules. It must be noted that the output from the Gazebo topic comes in quaternions, so a direct relay of orientation is not possible, because the rest of the system is euclidean. So, a transformation was done before exposing orientation to the rest of the system.

Closed loop with position control This stage is the first to include both open and closed loops. The open loop stays the same with the controller. Additionally, three more new modules create the closed loop. One of them is a reference coordinate, which can be static or dynamically recorded. This is the reference serves as the input for the control loop. The second new module is an error computation operator. It reads the position from the state estimator and compares it to the reference position. The

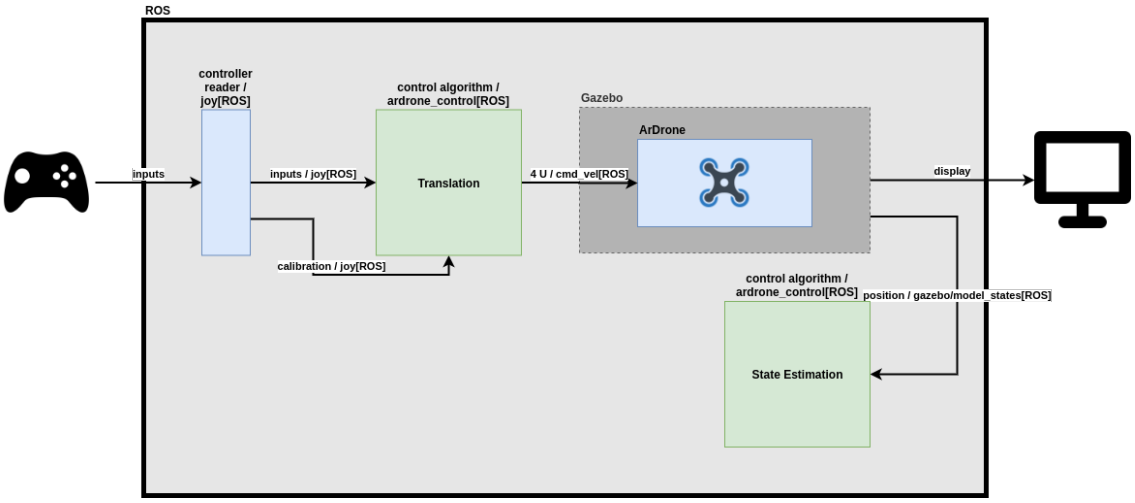

Figure 18: Open loop model with position sensing, and velocity and acceleration estimation.

difference (error) is then fed to the control aggregator, who merges this error with the user inputs and finally publishes it to the command topic. Thereby, we can consider the loop closed.

As it can be deduced, at this point there is no system controller present, meaning that the position-to-command conversion is one to one (1:1 or $k = 1$). It does not make a big difference. The only change is just that the proportional gain may not be adequate for the system, but is negligible due to the integrative nature of the target system.

As we can also see, the control system shifted from a velocity approach model to a position approach model. This necessary, because position is the base output from the system, and the best observable. Also, the closed loop is not always active. It can be manually enabled by the user (see enable tag in error operator).

Fault detection Two new modules were added to this stage. The PID controller module is one. Its task is to modify the response coming from the error and pass that modified input to the aggregator. The drone now has a softened and more precise response to position control. It no longer oscillates as much near the reference.

The other implemented module is the fault detection module. The method chosen was to compare the velocity command coming from the user against the estimated velocity of the drone. A threshold was set for the difference. So, each time the module computes an overshot difference, a possible fault state is assumed. If the state is the same after several iterations, the module triggers an alarm response. In this configuration, the trigger only acts on the error module, so it closes the loop.

The behavior of this stage is in this manner: the user pilots the drone as an open loop control. The system is supervising the flight at all times. As soon as the fault detection module declares a fault, it triggers the closed loop control, and resets the

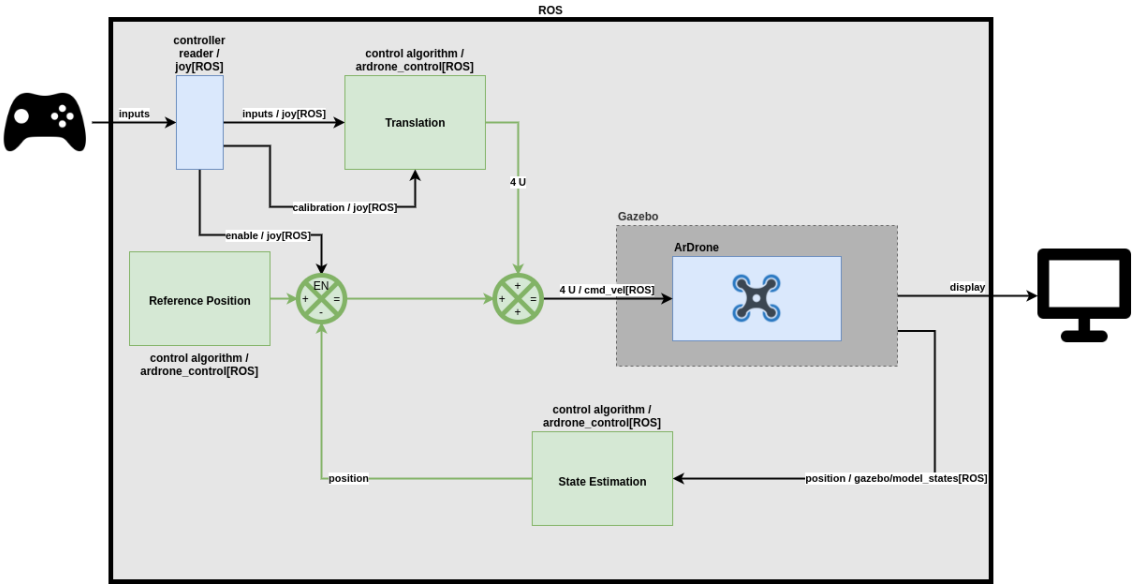

Figure 19: Closed loop positional model with external velocity control.

drone to a central position.

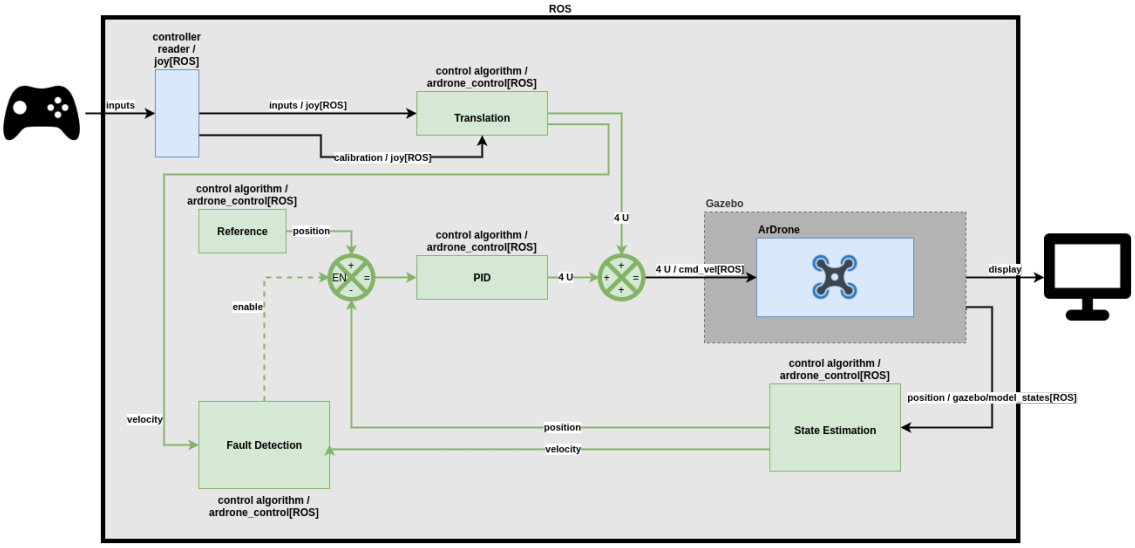

Figure 20: Closed loop detection.

FTC This is the final stage, the one culminating the project. It just adds a single module, but the most important one because it brings all of it together. This module is capable of compensating faults, whenever a fault is declared. The procedure when a fault appears is now as follows: when the fault detection module enables the closed loop, this module is also enabled. Then the drone is brought to a certain position,

either immediately recorded when the fault is declared or previously set. After a while, the drone is freed to continue its flight, without any observable fault. Just like that, the fault is suppressed.

Internally, the fault correction module is always observing velocity commands, reference positions and estimated positions. Also, the initial command values sent to the aggregator are zero. Whenever a fault is declared, this module observes the drone trying to be stabilized. Once the quadcopter is brought near enough to a stable flight, this module waits a while. If the module considers it has enough stability, it releases all closed loop enables and immediately records the current published commands. These recorded commands become now the output commands passed to the aggregator. These values compensate directly the fault's effect on the drone, restoring back control to its desired state.

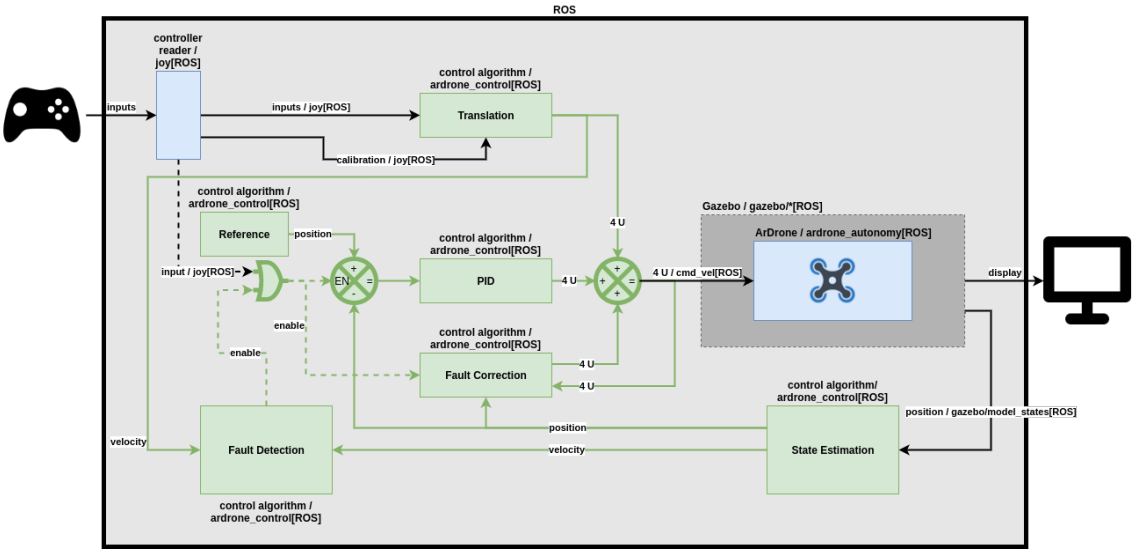

Figure 21: Closed loop detection and correction.

4 Experimental Results

The entire control system was tested inside the simulation environment, as previously described. Additionally, a small random fault generator module was added to the algorithm, as a method of creating aleatory faults in the UAV. The specific faults we are generating are additional inputs. However, with this design, the system is capable of recovering from faults adding to the actuator inputs, and the controlled system itself. The way it acts is by responding to a specific button press in the user's controller, and proceeds to add a pseudo-random amount of deviation to each control input. That fault remains until fault correction is reset by the user.

To have better visibility into the experimental part of the project, only one variable was recorded and documented. In this case, linear X axis was chosen for testing, which

is the forwards/backwards movement. However, the system is completely capable of correcting faults in the four control axis simultaneously. Finally, all measurements presented here were directly logged by the algorithm, and saved into a data file. All data was graphed afterwards without modification, just adjusting the window to a smaller size due to the increased amount of data generated.

Please note the recording format is in international units and saved each iteration of the algorithm. No time is presented due to the nature of the system (ROS node), which is to trigger at uneven intervals, relative to the simulator. Because the simulator is in a computer thread separate from the ROS node, processing is not synchronized. Therefore, times between programs drift. As a result, it is imprecise to mark saved times in with time, since times are not coordinated. Also, we must consider both simulation time and node time lag behind real time. However, for reference, the rate of the ROS algorithm is capped at a maximum of 50 iterations per real second. So, roughly, every 49 or 50 iterations can be counted as a real second. So, in total, the captured interval has an estimate equivalent length of 32 real seconds.

With not much further to say, these are the results.

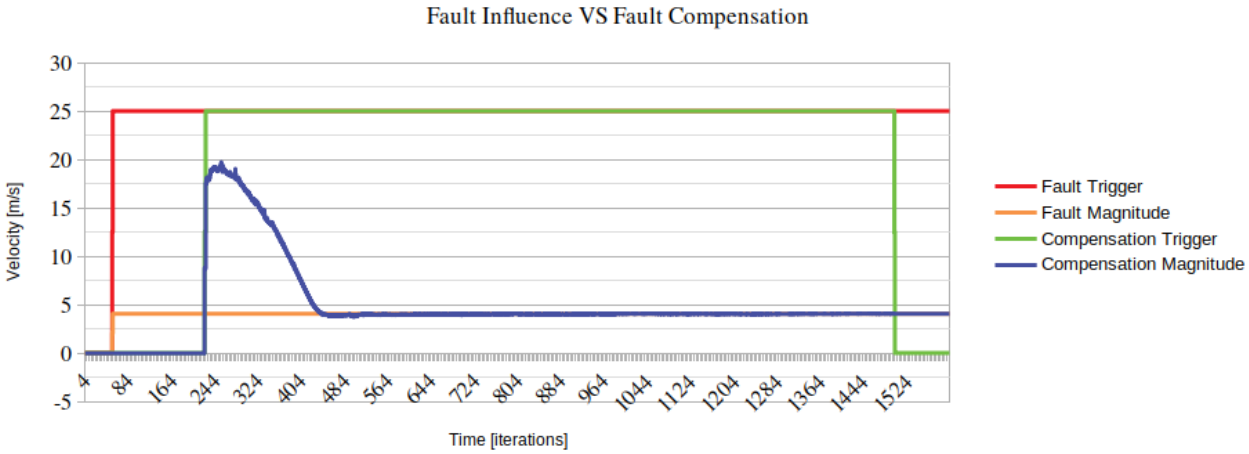

Figure 22: Graph showing triggers, and fault and compensation commands.

First, we observe at the very beginning, the nominal state for the drone. We read it is hovering in place near X coordinate zero, without any trigger going off. This is the ideal, non-faulty state which can be observed across all graphs.

However, around iteration 80, a fault appears. We know that by looking at the trigger (red line, Figure 22). It engages, and its magnitude rises from zero to about $4m/s$ (light orange line, Figure 22). Almost immediately, velocity rises proportionally, sliding to the negatives (red line, Figure 23). As an effect, the quadcopter starts moving backwards because of said velocity (black line, Figure 24). This behavior is characteristic of a fault. The system is operating normally, and suddenly, something changes somewhere in the system and starts driving said system out of control. In this case,

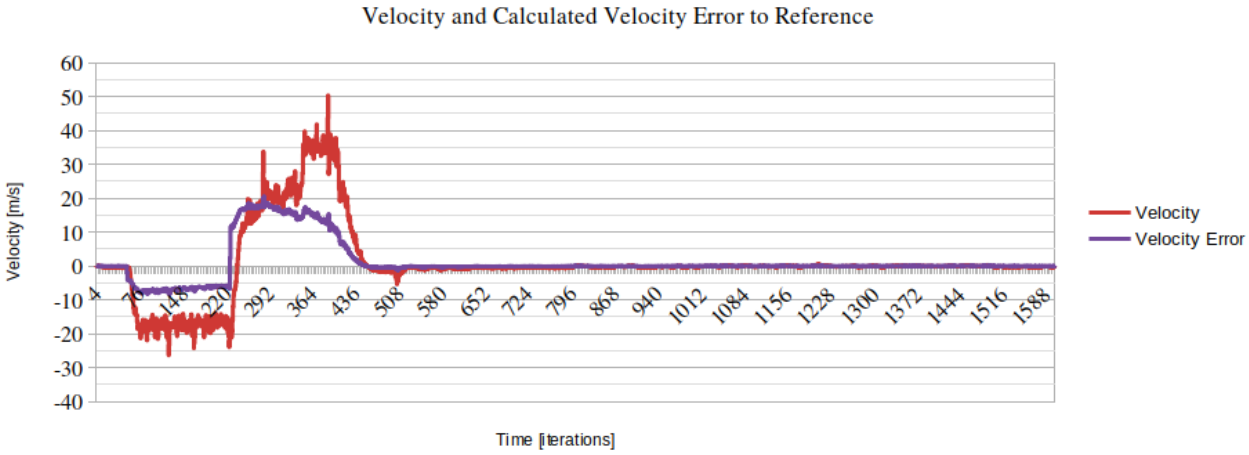

Figure 23: Graph showing estimated velocity and calculated error.

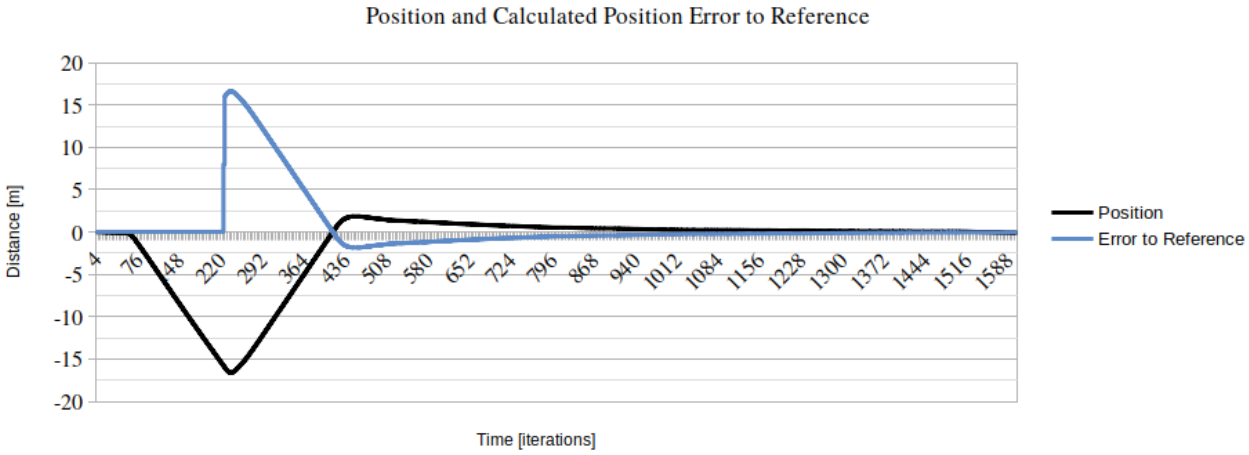

Figure 24: Graph showing linear X axis position and calculated error.

the UAV starts flying away increasingly faster each time.

During this period of time, the fault detector module is registering anomaly after anomaly. This can be seen in the graph where velocity error appears. Because of the unexpected behavior, the error becomes a non-zero value (purple line, Figure 23), and the recurrent fault state is noted. After iteration 240, the module decides this is an official fault state, and enables the closed loop state. There is a second trigger appearing, which is the compensation trigger (green line, Figure 22).

So, all correction mechanisms activate. If we focus on the positional error, we see that it is at that exact moment of the trigger when the loop starts calculating the error (light blue line, Figure 24). The position reference enters into play, and all control is assumed by the system. The PID does its job nicely, as observed in the position curve

(Figure 24). It eventually reaches an static state without much overshoot. The error gets reduced to zero in about 1000 iterations. The control command (compensation magnitude) also reaches a flat state fairly quickly (dark blue line, Figure 22).

In overview, the system regains control of the aircraft and returns it to the desired state. For some more iterations, the fault correction module lets the system further reduce the error to almost absolute zero. Finally, after iteration 1500, the module assumes it has reached the recovered state and disables the closed loop control (green line, Figure 22). Nonetheless, compensation stays the same, matching fault magnitude (dark blue line, Figure 22). As a result, the fault is suppressed and the drone keeps flying nominally without further incidents derived from this fault.

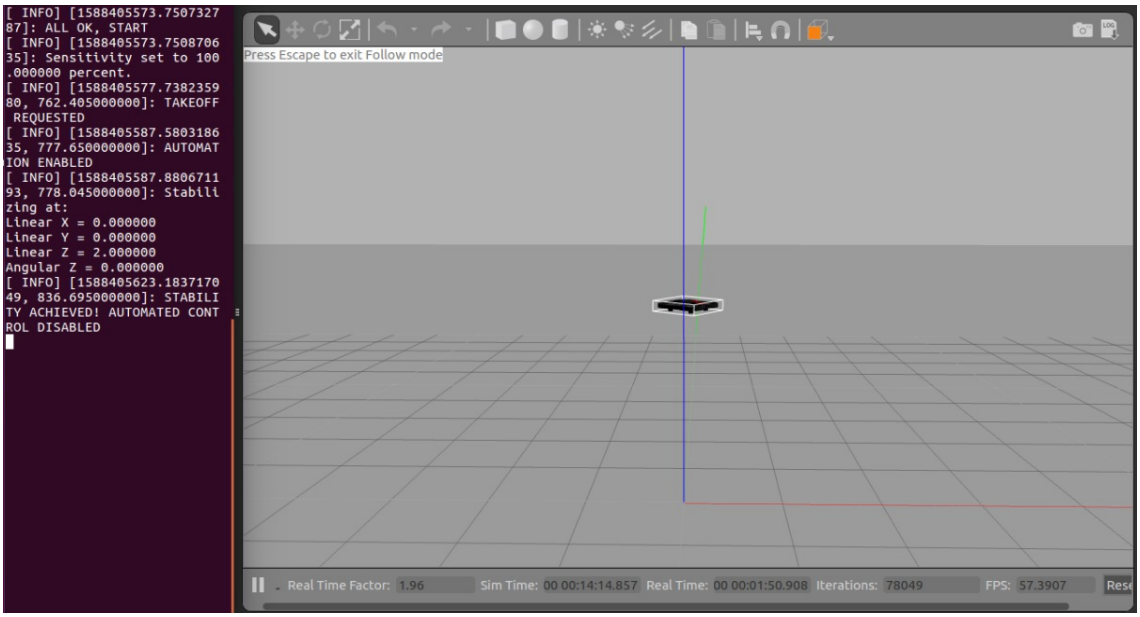

Figure 25: Simulation environment used in conjunction with ROS.

5 Conclusions

This work has demonstrated that commercially available drones can be enhanced and made better, despite with their limited characteristics. With just a single control engineering principle applied to a drone, we can ensure a more stable flight. And even when a fault condition appears, the drone remains safely and reliably in flight, ready for user operation. This type of ideology should be adopted by drone designers, so interest in drones can be regained, and further adoption of this type of vehicles is possible.

Talking in a more technical language, we achieved to integrate an advanced control technique to a UAV. Something that is usually reserved for industrial high value equipment was translated to a different playing field. Although it was not a simple solution,

the integration of a fault tolerant control proved to be enough to maintain this system in flight. And, control was restored successfully.

Future work on this project should be done in the fault detection algorithm or the state estimator. Sometimes, when testing, a fault would get detected even when the actual fault was not in effect. In other words, a false positive would appear due to the great noise coming out of the estimator. It may be related to the simple filtering technique used to read the signal, making the system prone to false positives. My approach to fix that was to increment greatly the threshold, giving the algorithm more space to spike and not trigger false positives. So, seeing this behavior, is why I think more work should be done in that part.

The direction I recommend to take is optimizing the estimator, or implement a better filtering technique. Since the filter comes right after the estimator, modifying either of those would make the project more reliable. Velocity and acceleration are calculated using simple delta time divisions (time steps), so if the process lags a little, due to the time separation being so small (about one twentieth of a second), results vary widely from sample to sample. That is why we see so any spikes in the chats portraying estimated velocity. So, the estimator could use a different derivation method. Alternatively, the filter could receive the rework. It currently remembers the last 6 steps calculated in the estimator (about a third of a second worth of samples) and performs a simple averaging method. The results then get passed on to the trigger mechanism. If a better filtering technique is used on the estimated velocities, the threshold will then be hit only when a real fault occurs. These two parts are the ones that need more work and, if fixed, would certainly make the project produce more polished results.

Finally, the help of the ROS platform proved to be a great accelerator for the project, and in difficult times like this one, a simulator such as Gazebo comes in very handy. The simulator played such an important role that it was not critical to be in a real testing facility, even if the algorithm is intended to be used in a real testing environment. We must give enough credit to the open source initiative, which made this project possible.

6 References

[Belikovetsky et al., 2016] Belikovetsky, S., Yampolskiy, M., Toh, J., and Elovici, Y. (2016). dr0wned - cyber-physical attack with additive manufacturing.

[Bergin, 2014] Bergin, C. (2014). Spacex's autonomous spaceport drone ship ready for action. nasaspaceflight.com/2014/11/spacex-autonomous-spaceport-drone-ship/.

[Blanke et al., 2000] Blanke, M., Frei], W. C., Kraus, F., Patton], J. R., and Staroswiecki, M. (2000). What is fault-tolerant control? *IFAC Proceedings Volumes*, 33(11):41 – 52. 4th IFAC Symposium on Fault Detection, Supervision and Safety for Technical Processes 2000 (SAFEPROCESS 2000), Budapest, Hungary, 14-16 June 2000.

[COMUNICACIONES Y TRANSPORTES, 2019] COMUNICACIONES Y TRANSPORTES, S. (2019). Norma oficial mexicana nom-107-sct3-2019, que establece los requerimientos para operar un sistema de aeronave pilotada a distancia (rpas) en el espacio aéreo mexicano.

[Darío Maravall and Fuentes, 2017] Darío Maravall, J. d. L. and Fuentes, J. P. (2017). Navigation and self-semantic location of drones in indoor environments by combining the visual bug algorithm and entropy-based vision.

[Digital Trends, 2018] Digital Trends (2018). The history of drones in 10 milestones. https://www.digitaltrends.com/cool-tech/history-of-drones/.

[Dijkshoorn and Visser, 2011] Dijkshoorn, N. and Visser, A. (2011). Integrating sensor and motion models to localize an autonomous ar. drone. *International Journal of Micro Air Vehicles*, 3:183–200.

[eborghi10, 2016] eborghi10 (2016). Ar.drone-ros. https://github.com/eborghi10/AR.Drone-ROS.

[Edward F. Hogge and Quach, 2015] Edward F. Hogge, Brian M. Bole, S. L. V. J. R. C. T. H. S. B. L. H. K. M. S. and Quach, C. C. (2015). Verification of a remaining flying time prediction system for small electric aircraft. http://hdl.handle.net/2060/20160006424.

[Eschmann et al., 2013] Eschmann, C., Kuo, C.-M., Kuo, C.-H., and Boller, C. (2013). High-resolution multisensor infrastructure inspection with unmanned aircraft systems. *ISPRS - International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, XL-1/W2:125–129.

[Farahmand, 2020] Farahmand, F. (2020). Circuit switching circuit switching. https://www.researchgate.net/publication/242401181.

[Franklin, 2002] Franklin, G. F. (2002). *Feedback Control of Dynamic Systems*.

[Furrer et al., 2016] Furrer, F., Burri, M., Achtelik, M., and Siegwart, R. (2016). *RotorS – A Modular Gazebo MAV Simulator Framework*, volume 625, pages 595–625.

[Hall, 2015a] Hall, N. (2015a). The lift equation. https://www.grc.nasa.gov/WWW/k-12/airplane/lifteq.html.

[Hall, 2015b] Hall, N. (2015b). Newton's laws of motion. https://www.grc.nasa.gov/WWW/K-12/airplane/newton.html.

[Huang and Sturm, 2018] Huang, H. and Sturm, J. (2018). tum_simulator. http://wiki.ros.org/tum_simulator.

[Kaufmann et al., 2019] Kaufmann, E., Gehrig, M., Foehn, P., Ranftl, R., Dosovitskiy, A., Koltun, V., and Scaramuzza, D. (2019). Beauty and the beast: Optimal methods meet learning for drone racing. pages 690–696.

[Monajjemi, 2015] Monajjemi, M. (2015). ardrone_autonomy. http://wiki.ros.org/ardrone_autonomy.

[O'Donnell, 2019] O'Donnell, S. (2019). A short history of unmanned aerial vehicles. https://consortiq.com/en-gb/media-centre/blog/short-history-unmanned-aerial-vehicles-uavs.

[Olejnik, 2016] Olejnik, A. (2016). Trends in the development of unmanned marine technology. *Polish Hyperbaric Research*, 55.

[Open Source Robotics Foundation, 2014] Open Source Robotics Foundation (2014). Gazebo components. http://gazebosim.org/tutorials?tut=components&cat=get_started.

[Pothuganti et al., 2017] Pothuganti, K., Jariso, M., and Kale, P. (2017). A review on geo mapping with unmanned aerial vehicles. *International Journal of Innovative Research in Computer and Communication Engineering*, 3297.

[ROS.org, 2011] ROS.org (2011). joystick_drivers. http://wiki.ros.org/joystick_drivers.

[ROS.org, 2016] ROS.org (2016). Ros kinetic kame. http://wiki.ros.org/kinetic.

[ROS.org, 2018] ROS.org (2018). Documentation. http://wiki.ros.org/Documentation.

[SpaceX, 2020] SpaceX (2020). https://www.spacex.com/dragon.

[The Slow Mo Guys, 2015] The Slow Mo Guys (2015). Cd shattering at 170,000fps! - the slow mo guys. https://youtu.be/zs7x1Hu29Wc.

[Verde, 2013] Verde, C. (2013). *Monitoreo y diagnoóstico automático de fallas en sistemas dinámicos*. Trillas Instituto de ingeniería UNAM, México, D.F.

[Villegas et al., 2012] Villegas, N., Tamura, G., Müller, H., Duchien, L., and Casallas, R. (2012). *DYNAMICO: A Reference Model for Governing Control Objectives and Context Relevance in Self-Adaptive Software Systems*, volume 7475.

[Väljaots and Sell, 2019] Väljaots, E. and Sell, R. (2019). Energy efficiency profiles for unmanned ground vehicles. *Proceedings of the Estonian Academy of Sciences*, 68:55.

[Wilcher, 2019] Wilcher, D. (2019). Ros 101: An intro to the robot operating system. https://www.designnews.com/gadget-freak/ros-101-intro-robot-operating-system/107053141061075.

May 16th, 2020