CENTRO DE INVESTIGACIÓN Y DE ESTUDIOS AVANZADOS DEL INSTITUTO POLITÉCNICO NACIONAL

UNIDAD ZACATENCO

PROGRAMA DE SISTEMAS AUTÓNOMOS DE NAVEGACION AÉREA Y SUBMARINA

"Mapeo y Localización Simultáneos en una Plataforma Cuadrirotor"

T E S I S

Que presenta

Jossué Cariño Escobar

Para obtener el grado de

MAESTRO EN CIENCIAS

EN SISTEMAS AUTÓNOMOS DE NAVEGACIÓN AÉREA Y SUBMARINA

Directores de la Tesis:

Dr. Pedro Castillo García

Dr. Sergio Salazar Cruz

México, D.F. Abril, 2015

CENTRO DE INVESTIGACIÓN Y DE ESTUDIOS AVANZADOS DEL INSTITUTO POLITÉCNICO NACIONAL

UNIDAD ZACATENCO

PROGRAMA DE SISTEMAS AUTÓNOMOS DE NAVEGACION AÉREA Y SUBMARINA

"Simultaneous Localization and Mapping on a Quadrotor Platform."

T H E S I S

Presented by

JOSSUÉ CARIÑO ESCOBAR

Submitted in fulfillment of the degree of

MASTER OF SCIENCE

IN

SISTEMAS AUTÓNOMOS DE NAVEGACIÓN AÉREA Y SUBMARINA

Thesis supervisors:

PHD. PEDRO CASTILLO GARCÍA

PHD. SERGIO SALAZAR CRUZ

México, D.F. April, 2015

"'Do you know, I always thought Unicorns were fabulous monsters, too? I never saw one alive before!'
'Well, now that we have seen each other' said the unicorn, 'if you'll believe in me, I'll believe in you.'"

Lewis Carroll, Through The Looking Glass

Agradecimientos

Quisiera empezar por agradecer a todas las instituciones que hicieron posible culminar este trabajo. Agradezco al Consejo Nacional de Ciencia y Tecnología (CONACYT),por medio de su programa de becas, el haberme brindado el apoyo económico necesario para poder concluir mis estudios de maestría de forma satisfactoria y haber brindado el apoyo para poder realizar una estancia de investigación en el Francia. También le agradezco al Centro de Investigación y Estudios Avanzados del Instituto Politécnico Nacional (CINVESTAV)

Dedicatoria

A todos aquellos que siguen creyendo en unicornios y dragones gracias al niño/niña que llevamos dentro.
A mis padres, por haber inculcado en mi la semilla que en estos momentos esta dando frutos.
A mis hermanos, que siempre me impulsaron a sacar lo mejor de mi persona.
A mi familia, por brindarme una perspectiva tan bella y hermosa del mundo.
A mi amada María Fernanda Martínez Fernández de Castro, por haber estado en cada momento no sólo apoyándome sino también impulsándome hacía cumplir nuestros sueños.

Contents

List of Figures

List of Tables

Abbreviations

DoF	**D**egree **o**f **F**reedom
UAV	**U**nmanned **A**erial **V**ehicle
SLAM	**S**imultaneous **L**ocalization **A**nd **M**apping
VTOL	**V**ertical **T**ake-off and **L**anding
RANSAC	**RAN**dom **SA**mple **C**onsensus
KF	**K**alman **F**ilter
EKF	**E**xtended **K**alman **F**ilter
ROS	**R**obot **O**perating **S**ystem
RF	**R**adio **F**requency
3D	**3**-**D**imensional
2D	**2**-**D**imensional

Resumen(English)

Unmanned aerial vehicles are starting to have a greater impact both on commercial and in the robotic applications. The technology nowadays has made it possible to produce aircrafts with low-cost, high processing power and longer battery duration. In particular, inertial measurement units have decreased their cost considerably by using more efficient algorithms to filter inexpensive sensors, thus making it easier to stabilize a vehicle's attitude. It has also been possible for multirotor platforms to stabilize their velocity by using optical flow and vision algorithms. However, even now there are few commercial sensors that measure an aircraft system's position that can be put inside these platforms, besides the GPS systems.

The Quadrotor platform has proven to be able to maneuver in reduced and indoors environments, but requires its position in order to be stable. GPS signal is usually very poor inside buildings and urban corridors, so another method is needed for localizing the drone. Also, the vehicle can bump or even crash with the surroundings if they are not properly mapped. Solving this simultaneous localization and mapping problem, or SLAM, under this conditions is a necessary step in order to increase this platform's applications.

In this work the SLAM problem is addressed using a laser range finder sensor to map and locate a drone in an unstructured environment using a quaternion system model. The main advantage of this scheme is that the vehicle's model can be seen as an axis-angle linear representation that reduces the complexity of the calculations. The algorithm used in this case is the extended Kalman filter, but this concept can also be applied in other solutions of the SLAM problem.

Keywords. SLAM, EKF, KF, ROS, Quaternion, Dynamics, Hypercomplex numbers, Sliding mode, UAV, Drones, Multirotor

Resumen

Los vehículos aéreos no tripulados estan empezando a tener un mayor impacto en aplicaciones comerciales y robóticas. Las tecnologías emergentes han permitido producir aeronaves con un coste bajo de producción, alto poder de procesamiento y una mayor duración de las baterías. En particular, se ha reducido considerablemente el coste de fabricación de las centrales inerciales al usar algoritmos más eficientes para filtrar datos de sensores de bajo costo, lo que se traduce en que sea más fácil estabilizar la orientación de vehículos aéreos. También ha sido posible para plataformas multirotoras estabilizar su velocidad traslacional al utilizar sensores de flujo óptico y algoritmos de visión. Sin embargo, en la actualidad existen pocos sensores comerciales que puedan medir la posición de un vehículo y que sean capaces de integrarse adentro de estas plataformas, aparte del sistema de posicionamiento global GPS.

La plataforma cuadrirotor ha demostrado ser capaz de maniobrar en espacios reducidos y hambientes cerrados, pero requiere conocer su posición aproximada para ser estable. La señal de los sensores de GPS normalmente es muy débil dentro de edificios y en corredores urbanos, por lo que es necesario otro método para localizar al drone. También, el vehículo puede llegar a chocar o incluso estrellarse con el propio ambiente si es que no se tiene un mapa adecuado del mismo. Resolver este problema de mapeo y localización simultáneos, o *SLAM* por sus siglas en inglés, propiciará el desarrollo de mayores aplicaciones para la plataforma cuadrirotor.

En este trabajo el problema de SLAM se ataca usando un sensor de barrido láser para mapear y localizar al drone en ambientes no estructurados usando un modelo del sistema basado en cuaterniones. La principal ventaja de este esquema es que el modelo del vehículo se puede ver analizar como un sistema linear en la representación eje-ángulo que reduce la complejidad del diseño de algoritmos de control y de observación. El algorimto empleado en este caso para resolver el problema de SLAM se basa en el filtro extendido de Kalman, pero este esquema se puede aplicar también a otros algoritmos de SLAM.

Keywords. SLAM, EKF, KF, ROS, Quaternion, Dynamics, Hypercomplex numbers, Sliding mode, UAV, Drones, Multirotor

Chapter 1

Introduction

Unmanned aerial vehicles, commonly referred as UAVs, are a relatively new array of aerial robotic platforms that are becoming more common. They offer more flexibility in terms of speed and maneuverability in contrast to ground and underwater platforms. The current technology enables them to become fully autonomous for many tasks such as following paths, surveillance and aerial video capture. For this reason there has been an increasing interest in using them as vehicle platforms. The increasing popularity of UAV comes from the fact that the cost of producing and testing them has reduced drastically in recent years. Amongst the many types of aerial platforms, one of the most studied and used because of its simple model is the quadcopter, which is a special form of multicopter.

Current research in UAV navigation focuses primarily on problems associated with the interactions between the UAV and its surrounding environment. The main advantage of aerial vehicle is their mobility, but it can be affected by external factors that are difficult to predict and compensate. The focus of this work is on one of the most fundamental aspects of the vehicle, its position.

As drones move through their six degrees of freedom, they need to be aware of their surroundings in order to guarantee a safe flight. The environment can consist of buildings, the ground, possible obstacles, even other drones. However, the real challenge is how to know the location of all these features in a static frame of reference because the majority of sensors used in UAVs are relative to other types of frames and rarely give a measurement in a global frame. One of the few sensors that can locate an airborne drone is a GPS sensor. This sensor has good accuracy and response time in open areas, but it can cause the UAV to become unstable if used indoors or in closed environments because they have poor satellite reception.

Stability indoors or in areas with poor reception can be accomplished using a down-facing camera that captures the horizontal velocity of the vehicle using an optical flow algorithm. This method is effective to make the vehicle stay in one place, but it cannot estimate accurately the position of the vehicle when following a path because of the inherent drift due to the sensor's noise. Another problem that arises when trying to navigate is detecting and avoiding any obstacles that may be present among the vehicle's path. The problem involving the characterization of the environment is referred to as mapping, while the one of determining the pose of an UAV is addressed as localization. The problem of locating and mapping simultaneously is referred to as SLAM and it is a very common topic in robotic research as both of them are inherently linked.

This work presents a solution to the SLAM problem using a quadcopter platform in unstructured environments. The sensors used in the prototype to test the SLAM algorithm were a laser range-finder sensor, an inertial measurement unit (IMU) with ten degrees of freedom (accelerometer, gyroscope, magnetometer and barometer) and an optical flow sensor. The map obtained from the algorithm can be used to avoid obstacles and, together with the position, for path planning and following.

Objectives There are many solutions to the SLAM problem in the current literature, as it is a well defined and common problem. The reason for delving deeper in this particular subject is that more extensive research in localization and mapping aids in making UAVs become a safer and more reliable platform. Also, SLAM solutions are usually applied for 2D environments, while the quadcopter is a platform that can perform this task in a 3D space.

One of the most used solution to the SLAM problem is the Extended Kalman Filter, or EKF, because of its simpleness and elegant solution. It has become one of the default standards when evaluating a SLAM algorithm. One of its fundamental weaknesses comes from the fact that the Kalman filter is designed to work with linear systems and additional complexity and operations need to be used in order to adapt it to non-linear systems and measurements. The SLAM solution presented in this work uses a linear axis-angle model and an analogous quaternion model for the attitude estimation. The use of these models reduces the complexity of the EKF and of the operations used, making its implementation more efficient. These results are tested on the quadrotor platform to obtain a representation of a 3D environment.

Chapter 2

Background and Related Works

In this chapter a review of the themes presented in this thesis are introduced, as well as a presentation of the state of the art of UAVs and SLAM algorithms. It is presented in order to show the basis on which the current work was developed and as a reference for future works on related subjects.

2.1 Unmanned Aerial Vehicles

The most important characteristic of UAVs is that they don't require the presence of an on-board pilot in order to fly properly. Many definitions state UAVs as aerial vehicles with this sole quality Nex and Remondino [2014]. For many UAV systems the absence of a pilot seems to be the only common feature, as there are a wide variety of forms and applications in which these vehicles can operate on this manner.

Even this attribute can be cryptic because there exist systems that can be controlled remotely without the need of any person on-board the vehicle using a Ground Control station (GCS) or any other kind of teleoperation, but the operator is a compulsory requirement. Some drone configurations still need a certain degree of autonomy from the pilot in order to even be stable, so another difference between them comes from how much autonomy they have in the sense of what kind of orders or references they receive from a GCS. In light of this, the most general classification of UAVs is based on the aerodynamic configuration used, which can either be a fixed-wing aircraft or a rotary wing model. There exists more configurations than the ones specified here, but these are the most widely used in the UAV field.

Fixed-Wing

Fixed-wing aircraft's most remarkable characteristic is that lift is achieved as a consequence of the airflow passing over one or more wings that are fixed to the body of the vehicle, hence the name. The main control input of these types of aircraft is on the various motors that normally generate a forward thrust, and the aerodynamic control surfaces that are attached to the aircraft.

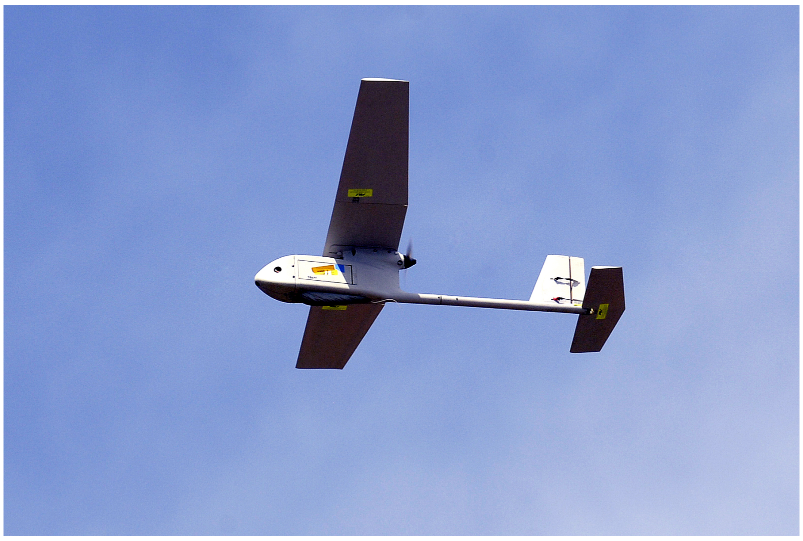

FIGURE 2.1: An example of a fixed-wing aircraft. Picture taken from [Force, U.S. Air Force]

These types of aircraft are very efficient in terms of energy consumption and are more robust to external perturbations. However, as the lift is dependent of the forward thrust, this means that this type of model can't maintain sustenance at low speeds and can't hover. An example can be seen in Figure 2.1.

Rotary Wing

Rotary wing aircrafts, unlike fixed-wing, create lift with the motion of a propeller that has a fixed axis of rotation. Normally, the orientation of the thrust generated by this rotational motion is used for both lift and propulsion, and doesn't require forward motion like in the case of the fixed-wing. They have more maneuverability than fixed-wing vehicles, but at the price of more energy consumption and slower speeds.

These aircrafts are classified according to the location on the vehicle's fixed frame and the quantity of rotary wings. The most common rotary model among manned aircrafts is the classical helicopter design that uses a single rotor to achieve but lift and thrust, and has another rotor wing for control and balance purposes. There exists other designs that use rotary wings like Vertical Take-off and Landing (VTOL) vehicles, but the focus nowadays is on multirotor designs.

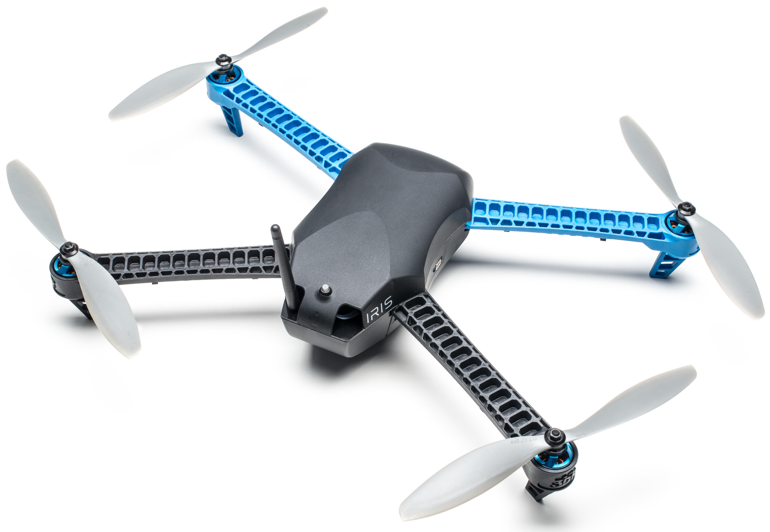

FIGURE 2.2: An example of a multirotor. Picture taken from [Industries, Adafruit Industries]

Multicopters, or multirotors, are a type of rotary wing aircraft that has more than one rotor in various configurations. The most common type of multirotors are the ones that have an even amount of motors with parallel thrust and counter-balancing rotary directions. The most simple example of this is the quadcopter, which consists of a frame structure with four motors placed in a square shape around it. The actuators are arranged so that two of them rotate clockwise and the other spin counter-clockwise to control the total torque generated by the motors. Other configurations come in different shapes, like the hexacopter and the octacopter which offer redundancy and a more stable platform, but the most simple is the quadcopter. An example of the latter can be seen in Figure 2.2.

Multirotors can move in all six DoF of a 3D environment, but in general they can only remain stable in four DoF (three DoF in position and one in the yaw angle). This is because the movements in pitch and roll are coupled with the platform's position.

The flexibility, low-cost and easy configuration of multicopters has made them the focus of research in the UAV field. They offer a very solid platform from which various applications have been developed and deployed, such as surveillance and photography, and other tasks are in the process of research.

Current Research Topics

Research in UAVs is primarily focused in the development of new kinds of configurations that can have the advantages of both kinds of platforms (rotary and fixed-wing), the research into new forms of control and stability and the research into applications that can be performed using UAVs. Below are listed some of the most prominent current applications and research found in Valavanis [2008].

Fault-tolerant flight A key feature of human pilots is their ability to adapt to drastic changes in-flight. They have the knowledge and experience to react to changes in the vehicle's model due to a fault. As most control strategies involve the knowledge of even part of the UAVs model, in the presence of a fault the control law may not be able to stabilize the system or can even make it unstable. Also, many UAVs configurations are not designed to be able to handle a fault, like a broken propeller or motor, as many don't have enough redundancy to compensate it.

A fault-tolerant flight is a combination of both a platform with redundancy together with an automatic control that can stabilize the system and maybe make it land whenever a fault presents itself. In Lanzon et al. [2014] a control law is designed for quadrotor vehicles which sacrifices the controllability of the yaw state due to rotor failure in order to allow the vehicle to use the remaining three functional rotors to stabilize the system.

Swarm operation One or more UAVs flying together with communication amongst them is called a swarm. The drones inside a swarm can cooperate to perform tasks that would be either too complex or too power or time consuming for one drone to do by itself. The challenge comes in the coordination of each of the individuals of the swarm, knowing where all the vehicles are at a given time, avoid collisions and planning the best path to accomplish the goal more efficiently. In Ma'sum et al. [2013], a swarm of UAVs is proposed for the localization and tracking of objects.

UAV Manipulator A manipulator is a kind of actuator that can manipulate objects. The most common form of manipulator is a gripper, which can take an object in order to translate it to a different pose. Most manipulators use a kinematic chain configuration. This type of operation is restricted to a finite area, whereas a manipulator mounted on an UAV has a wider work space that gives it flexibility. In Lee et al. [2013] a control law is proposed to take into account the added dynamical complexity of the arm.

Mapping A drone offers a very flexible platform in which to perform mapping, and the information can also be used by the vehicle. A map is a conception of the environment that has many uses for UAVs, like localization, collision detection and path planning. Mapping refers to the action of building a map from the information available from the UAVs sensors. The maps conceived by an UAV can be used in other applications, like geological map construction or 3D reconstruction of an environment. In Bryson et al. [2014] experimental results show that it was possible to produce geo-referenced maps for object detection and vegetation classification.

The problem of mapping is very intimately associated with the problem of localization in what many refer to in the literature as the SLAM problem. A solution to this is presented in more detain on chapter **??** , and a description of the problem can be encountered in the next section.

2.2 The SLAM Problem

In order to understand Simultaneous Localization and Mapping (SLAM), it is necessary to separate the problem in its two parts and then fuse them as one would usually find the situation in the application. The two parts involved are localization and mapping.

Localization in this context is the use of sensor data that is compared against an inner representation of the environment in order to find the best pose (translation and attitude) estimate of the moving platform. In this case, the inner map is extremely accurate and the sensor gives noisy data about the vehicle's surroundings, so the pose has to be filtered and estimated. Mapping is the action of building a representation of the vehicle's surroundings from sensor data using a very accurate estimate of the platform's pose. In this case, the noisy measurements have to be filtered with the aid of the accurate pose in order to scan the environment.

In practice, one cannot have an accurate pose estimate nor an exact map in memory, as the tool used for both instances is the same and has noise. Thus, the SLAM problem can be defines as the notion of trying to determine the pose and the environment by means of sensor data. Normally, this information comes from rangefinders or stereoscopic vision.

Applications of SLAM to UAVs As stated before, SLAM gives information about the environment and where the UAV is in it. The map can be used for trajectory planning, navigation, collision avoidance and 3D reconstruction. An advantage of doing the map on-line is that the drone can adapt to any kind of environment and even to changes on its surroundings, thus making the vehicle more robust and autonomous.

The UAV's position and attitude are mainly useful to help stabilize the drone in position. The most used sensor to determine a vehicles position is a GPS. This results inadequate for the stabilization of drones because of its relatively low accuracy. UAVs tend to have a very fast reaction time, and as such, they need accurate pose information or they could crash trying to stabilize themselves in position. SLAM can be used as an alternative to the GPS because it gives a more accurate pose estimate.

The main disadvantage of using SLAM on a drone is the velocity of the algorithm. Most SLAM implementations tend to be computationally intensive, because of the type of sensors used and the calculations involved to process the information and filter it. As of late, there has been a lot of interest on SLAM because the technology has made it possible to give UAVs enough computational resources to handle large quantities of information, making SLAM a more attractive option.

SLAM Diversity There are a lot of algorithms that solve the SLAM problem, each of which uses different methods, sensors and hypothesis as they tend to be tailored to very specific platforms and situations. However, the basic SLAM structure tends to remain the same.

Every SLAM algorithm needs, in one form or another, a way to store and update its map, a matching algorithm that can relate scanned data with the map, a pose calculation method and a filter algorithm that guarantees a stable estimation.

Map Creation and Search

There are various ways to represent a map. In SLAM, they can be divided in two: feature-based and grid-based.

Grid-based mapping stores the information by dividing the environment state in finite elements and assigning them to memory values. Each value in the map represents a finite volume in reality, and its value represents the state of that finite space in the surrounding such as occupied, or empty or even unknown. The advantage of using this method is that every finite space in memory directly relates to a physical object in the environment, which is easier to visualize. The main disadvantage is that it tends to consume more memory space, and thus resources, than other methods.

Feature-based mapping stores features. A feature is a collection of environment data that has a specific form, such as lines, planes, circles, etc. This kind of mapping consumes less memory space than grid-based methods, as it only stores the minimum data of a given feature. For example, if there was a line grid-based mapping would store every point of the line while feature-based would only store the nearest line point. The advantage of this method is that it consumes less resources than grid-based mapping. The disadvantage is that not all the environment information is used, thus reconstructing the surroundings tends to consume more resources.

Independently of the representation used, SLAM often requires through searches through all the map information in order to be able to match the data from the sensor with the information in the map. This process is often called matching.

Localization

The estimation of the vehicle's pose from sensor measurements is a non-trivial problem when dealing with a lot of information. The main approach to solve this is as an optimization problem. The idea is to find the position and attitude that when applied to the scanned environment points results in the best fit, that is, given a series of measurements l_i and their corresponding map feature m_i, find the pose $P(\cdot)$ that when applied to every l_i results in the closest m_i.

SLAM Methods

As stated before, there are many algorithms and implementations that solve the SLAM problem. The general idea behind a SLAM implementation has been presented before. The following part tries to resume the most used methods to solve the SLAM problem, but the execution varies from one platform to another. This list of methods was compiled from the work of Oliver [2015]

EKF

The EKF is an approach that:

> "Uses a series of measurements over time containing noises and inaccuracies and produces estimates of unknown variables that are more precise than when based on single measurements." Bajracharya [2014]

Techniques based on the Kalman filter are one of the most popular approaches to SLAM. They give a way to update the internal map at the same time that it updates the estimated pose in order to give the most statistically accurate representation of the system and the map. It can also fuse the data from the SLAM sensor in order to increase the accuracy, like the sensor from an IMU. It is a recursive algorithm that takes the latest measurements to update and improve the previous state.

The Kalman filter was originally intended for linear systems, but a variant known as the Extended Kalman Filter is more often used because it can be applied to non-linear systems. The principle is the same as the Kalman filter, but the process to be estimated is linearized in every step in order to account for the non-linearities. A more thorough introduction to the KF and EKF can be found in Welch and Bishop [2006]

The advantage of this filter is that it only needs the previous measurement and the current sensor scan to work at each iteration. The main drawback is that it consumes a lot of resources due to the matrix inverse, as well as having to update and maintain a covariance matrix.

Rao-Blackwellized Particle Filters

Rao-Blackwellized particle filters are a way to track multiple possible solutions to the pose and map estimates. At each time step, random possible solutions to the state vector called particles are generated using the system's stochastic model. As sensor scans arrive, the generated particles are compared with the measurement stochastic model. Particles with unlikely solutions are eliminated leaving only the most probable solutions on each sensor update. This process is repeated indefinitely, leaving an ever-decreasing amount of particles.

The advantage of this type of filter is that it becomes less computational expensive over time, and estimates the best choice. The main disadvantage is that there is no guarantee that the surviving particles are the global best, as the particles tend to generate randomly.

Chapter 3

Model

In this chapter the mathematical model of the quadrotor is obtained and presented. This model uses unit quaternions instead of the more commonly-used Euler angles to describe the platform's attitude. A brief description of quaternion algebra and its properties are presented in appendix A. Unit quaternions, the quaternion attitude dynamic representation, the quadcopter quaternion model and finally the quadrotor attitude control and simulation are presented and discussed next.

3.1 Unit Quaternions.

Euler Angles are an intuitive way to view rotations in three dimensional space. However, they are mathematically hard to work with. As the idea of an automatic control law often demands a simple mathematical solution for achieving high update speeds, Euler angles fall short in terms of computational simplicity compared to other methods.

A unit quaternion is a quaternion $\boldsymbol{q} \in \mathbb{H}$ whose norm is one $||\boldsymbol{q}|| = 1$. Unit quaternions are a subgroup of the quaternion group. It is often the preferred way to represent rotations in 3D space because of the advantages it has over other representations. These are:

- They use less data than rotation matrices (4 numbers instead of 9).
- They do not have singularities, as a product of a unit quaternion with any non-null vector will always fall inside the quaternion group.
- They do not have gimball-lock effect, as any attitude can be treated such that only one quaternion can define it. This is achieved by assuring that $q_0 \geq 0$.
- They are computationally more stable, as they can be normalized to reduce numeric errors before any operations.
- Their operations are computationally simple and efficient as the quaternion multiplication doesn't need trigonometric functions.

An interesting property of unit quaternions is that $\boldsymbol{q}^{-1} = \boldsymbol{q}^* \neq 0$. This can be seen from equation (A.8).

Euler-Rodríguez Equation of Rotation.

Euler's theorem for rigid bodies rotations implies that any rotation of a rigid body in 3D space can be viewed as a rotation by a certain amount or angle, with respect to a fixed axis that goes through the center (the center of mass in most cases) of a rigid body. A rotation in $\mathbb{R}^3$ space can be represented as:

$$
\begin{aligned}
&\boldsymbol{q} \in \mathbb{H}, ||\boldsymbol{q}|| = 1 \\
&\overline{p} \in \mathbb{R}^3 \\
&\overline{p}' = \boldsymbol{q}^{-1} \otimes \overline{p} \otimes \boldsymbol{q} = \boldsymbol{q}^* \otimes \overline{p} \otimes \boldsymbol{q} \\
&\boldsymbol{q} := \cos\left(\frac{\theta}{2}\right) + \overline{u}\sin\left(\frac{\theta}{2}\right)
\end{aligned}
\tag{3.1}
$$

where

- $\overline{p}$ is a 3D vector in the original reference frame.
- $\overline{p}'$ is the rotated vector $\overline{p}$ so that it is now in a new reference frame.

- $\overline{u} \in \mathbb{R}^3$ is a unit vector that represents the direction of the axis of rotation.
- θ defines the angle of rotation around the axis of rotation.

Equation (3.1) makes it possible to translate any vector from one reference frame into another.

From the double product, it can be seen that a quaternion $\boldsymbol{q}$ gives the same rotation as the quaternion $-\boldsymbol{q}$. In order to avoid this duality, we can add the inequality $q_0 \geq 0$, which assures us that each possible orientation of a rigid body can only relate to one element in the subgroup of unit quaternions.

Multiple Rotations.

When a series of rotations $\boldsymbol{q}_1, \boldsymbol{q}_2, \ldots, \boldsymbol{q}_n \in \mathbb{H}; ||\boldsymbol{q}_1|| = ||\boldsymbol{q}_2|| = \cdots = ||\boldsymbol{q}_n|| = 1$, is applied, one after the other, using the property described in equation (A.6) ($(\boldsymbol{q} \otimes \boldsymbol{r})^* = \boldsymbol{r}^* \otimes \boldsymbol{q}^*$) , to a vector we get the following:

$$\begin{aligned} \overline{p}' &= \boldsymbol{q}_n^* \otimes (\cdots \otimes (\boldsymbol{q}_2^* \otimes (\boldsymbol{q}_1^* \otimes \overline{p} \otimes \boldsymbol{q}_1) \otimes \boldsymbol{q}_2) \otimes \ldots) \otimes \boldsymbol{q}_n \\ &= (\boldsymbol{q}_n^* \otimes \cdots \otimes \boldsymbol{q}_1^*) \otimes \overline{p} \otimes (\boldsymbol{q}_1 \otimes \cdots \otimes \boldsymbol{q}_n) \\ &= \boldsymbol{q}^* \otimes \overline{p} \otimes \boldsymbol{q} \end{aligned} \tag{3.2}$$

This means that a series of rotations can be represented as a single unit quaternion $\boldsymbol{q} = \boldsymbol{q}_n \otimes \cdots \otimes \boldsymbol{q}_1$

3.1.1 Unit quaternion to other representations.

Unit quaternions can be viewed or described using other means to define rotations in 3D space. This is often useful as a more intuitive way to know the attitude of a vehicle at a certain point. The ones we will be addressing here are rotation matrix, Euler angles and axis-angle representations.

Rotation Matrix.

The quaternion rotation can be represented as a rotation matrix $\mathbb{R}^{3\times 3}$. This matrix can be constructed as:

$$Q(\boldsymbol{q}) = \begin{bmatrix} q_0^2+q_1^2-q_2^2-q_3^2 & 2(q_1q_2-q_0q_3) & 2(q_0q_2+q_1q_3) \\ 2(q_1q_2+q_0q_3) & q_0^2-q_1^2+q_2^2-q_3^2 & 2(q_2q_3+q_0q_1) \\ 2(q_1q_3-q_0q_2) & 2(q_0q_1+q_2q_3) & q_0^2-q_1^2-q_2^2+q_3^2 \end{bmatrix} \tag{3.3}$$

where $Q(\boldsymbol{q})$ is orthonormal, that is, $Q(\boldsymbol{q})^{-1} = Q(\boldsymbol{q})^T$ and $||Q(\boldsymbol{q})|| = 1$.

Euler Angles.

In order to represent quaternions in a more intuitive manner, they can be transformed into Euler angles. Fresk and Nikolakopoulos [2013] give the following formula in order to accomplish this:

$$\begin{bmatrix} \phi \\ \theta \\ \psi \end{bmatrix} = \begin{bmatrix} \arctan 2\left(2\left(q_0q_1+q_2q_3\right), q_0^2-q_1^2-q_2^2+q_3^2\right) \\ \arcsin\left(2\left(q_0q_2-q_3q_1\right)\right) \\ \arctan 2\left(2\left(q_0q_3+q_1q_2\right), q_0^2+q_1^2-q_2^2-q_3^2\right) \end{bmatrix} \tag{3.4}$$

Axis-Angle.

The Euler-Rodríguez formula (3.1) has a direct relationship between the axis of rotation, the angle of rotation and unit quaternions. However, it is possible to express the axis of rotation and the angle of rotation in a single vector. This representation is called **"Axis-Angle"**, and will be denoted as $\overline{\theta} \in \mathbb{R}^3$. The magnitude of this vector represents the angle of rotation used in (3.1), that is, $||\overline{\theta}|| = \theta$. The axis of rotation is obtained from the normalization of this vector, that is, $\overline{u} = \dfrac{\overline{\theta}}{||\overline{\theta}||}$.

To obtain a direct relationship between the axis-angle representation and a unit quaternion we use the quaternion logarithmic mapping. This mapping is defined as:

$$\begin{aligned} &\boldsymbol{q} \in \mathbb{H} \\ &\ln \boldsymbol{q} := \begin{cases} ||\overline{q}|| = 0, & \ln q_0 \\ ||\overline{q}|| \neq 0, & \ln ||\boldsymbol{q}|| + \frac{\overline{q}}{||\overline{q}||} \arccos \frac{q_0}{||\boldsymbol{q}||} \end{cases} \\ \\ &||\boldsymbol{q}|| = 1 \\ &\ln \boldsymbol{q} = \frac{\overline{q}}{||\overline{q}||} \arccos q_0 \end{aligned} \tag{3.5}$$

Using equation (3.5) and (3.1), we have the following axis-angle representation from an unit quaternion:

$$\begin{aligned} &\overline{\theta} \in \mathbb{R}^3; \boldsymbol{q} \in \mathbb{H} \\ &\overline{\theta} = 2 \ln \boldsymbol{q} \end{aligned} \tag{3.6}$$

where $\overline{\theta}$ is the axis-angle attitude and $\boldsymbol{q}$ denotes the attitude unit quaternion.

3.1.2 Unit quaternion from other representations.

Sensors measuring orientation, such as IMU's, may use other means to return attitude, such as Euler angles. In order to work using quaternions, some conversions must be defined.

Euler Angles.

Euler angles $(\boldsymbol{\phi}, \boldsymbol{\theta}, \boldsymbol{\psi})$, by definition, are three angles that define three rotations that give the attitude of a rigid body. These three rotations can be converted into the following quaternions:

$$\begin{aligned}
\boldsymbol{q}_\phi, \boldsymbol{q}_\theta, \boldsymbol{q}_\psi \in \mathbb{H} \quad & ||\boldsymbol{q}_\phi|| = ||\boldsymbol{q}_\theta|| = ||\boldsymbol{q}_\psi|| = 1 \\
\boldsymbol{q}_\phi = \quad & \cos\left(\frac{\boldsymbol{\phi}}{2}\right) + \sin\left(\frac{\boldsymbol{\phi}}{2}\right)\hat{i} \\
\boldsymbol{q}_\theta = \quad & \cos\left(\frac{\boldsymbol{\theta}}{2}\right) + \sin\left(\frac{\boldsymbol{\theta}}{2}\right)\hat{j} \\
\boldsymbol{q}_\psi = \quad & \cos\left(\frac{\boldsymbol{\psi}}{2}\right) + \sin\left(\frac{\boldsymbol{\psi}}{2}\right)\hat{k}
\end{aligned} \tag{3.7}$$

Using these quaternions, and the result of equation (3.2), the equivalent quaternion from Euler angles is:

$$\boldsymbol{q} = \boldsymbol{q}_\phi \otimes \boldsymbol{q}_\theta \otimes \boldsymbol{q}_\psi \tag{3.8}$$

Axis-Angle.

When using an Axis-angle representation, all the relevant information to use Euler-Rodríguez formula (3.1) is stored inside the axis-angle vector. In order to have a direct relationship between a unit quaternion and the axis-angle representation, we use the quaternion exponential mapping:

$$\boldsymbol{e}^{\boldsymbol{q}} := \begin{cases} ||\bar{q}|| = 0, & \boldsymbol{e}^{q_0} \\ ||\bar{q}|| \neq 0, & \boldsymbol{e}^{||\boldsymbol{q}||}\left(\cos\dfrac{||\boldsymbol{q}||}{2} + \dfrac{\bar{q}}{||\bar{q}||}\sin\dfrac{||\boldsymbol{q}||}{2}\right) \end{cases} \tag{3.9}$$

3.2 Quadcopter Dynamic Model.

A quadcopter is a kind of multirotor that has four actuators. It has four stable DoF, but can move in all six DoF. It is completely actuated in terms of attitude, and the translation can be changed by changing the orientation. The state vector can be defined as:

$$\begin{aligned} x_{quad} &:= \begin{bmatrix} x & y & z & \dot{x} & \dot{y} & \dot{z} & q_0 & q_1 & q_2 & q_3 & \omega_x & \omega_y & \omega_z \end{bmatrix}^T \\ &= \begin{bmatrix} \overline{p}^T & \dot{\overline{p}}^T & \boldsymbol{q}^T & \omega^T \end{bmatrix}^T \end{aligned} \tag{3.10}$$

where

- $\overline{p} = \begin{bmatrix} x & y & z \end{bmatrix}^T$ is the translational position vector with respect to the inertial frame.
- $\dot{\overline{p}} = \begin{bmatrix} \dot{x} & \dot{y} & \dot{z} \end{bmatrix}^T$ denotes the translational velocity vector with respect to the inertial frame.
- $\boldsymbol{q} = \begin{bmatrix} q_0 & q_1 & q_2 & q_3 \end{bmatrix}^T$ defines the vehicle orientation with respect to the inertial frame, represented as a unit quaternion.
- $\omega = \begin{bmatrix} \omega_x & \omega_y & \omega_z \end{bmatrix}^T$ represents the rotational velocity in the body frame.

In order to simplify the dynamic model, the quadcopter is divided into two subsystems, each one with their own dynamics and control law. The first subsystem is the orientation and attitude dynamics system. The second subsystem is used to control the position of the vehicle.

Quadcopter Forces. In the model it is assumed that only the motor forces act on the platform. Aerodynamic forces like wind and ground effect forces are assumed to be extremely small due to the relatively slow quadrotor's speeds and because an indoor environment is assumed. External forces like the manipulator arm's and gyroscopic forces are not included in this model, leaving only the effect on gravity on the platform.

The motor forces include the torques τ_i that each motor has on the platform, and the thrust that the motor's propellers generate, which is considered as:

$$f_i :\approx k_i \, \omega_i^2; \; i : 1, 2, 3, 4 \tag{3.11}$$

where ω_i is the motor angular speed and k_i means the motor thrust constant.

3.2.1 Attitude Dynamics.

Quaternions are used in the modeling and control of the quadcopter because they allow us to express the same non-linear mechanics, but in a linear way without loss of generality. In practice, the controller works using quaternions and quaternion operations. Figure 3.1 shows an image of the drone's free body diagram.

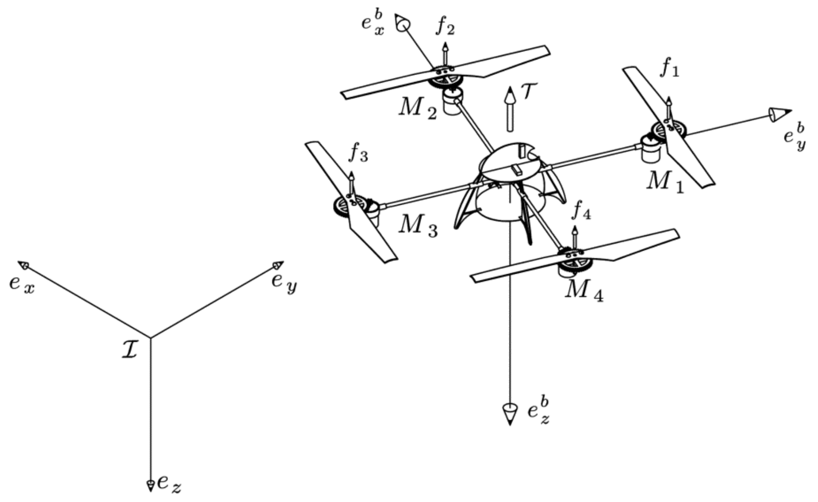

FIGURE 3.1: Quadrotor dynamic model free body diagram in NED reference frame. Image taken from Izaguirre-Espinosa [2015]

Quaternion Attitude Model. In order to describe a dynamic model using quaternions, we must first introduce the equation for quaternion kinematics for a rigid body:

$$
\begin{aligned}
&\omega \in \mathbb{R}^3,\ \boldsymbol{q} \in \mathbb{H} \\
&\dot{\boldsymbol{q}} := \frac{1}{2}\boldsymbol{q} \otimes \omega = \frac{1}{2} S\,\omega \\
&S := \begin{bmatrix} -q_1 & -q_2 & -q_3 \\ q_0 & -q_3 & q_2 \\ q_3 & q_0 & -q_1 \\ -q_2 & q_1 & q_0 \end{bmatrix}
\end{aligned}
\tag{3.12}
$$

where

- $\omega = [\ \omega_x \quad \omega_y \quad \omega_z\ ]^T$ is the angular velocity vector.
- $\boldsymbol{q}$ represents the attitude quaternion.

This allows us to express a quadrotor dynamic model in the following way:

$$
\begin{aligned}
&\omega, \tau \in \mathbb{R}^3;\ l = \sqrt{\frac{L}{2}},\ f_i, \tau_i, k_i, \omega_{M_i} \in \mathbb{R};\ \boldsymbol{q} \in \mathbb{H};\ i = 1,2,3,4 \\
f_i =&\ k_i\,\omega_{M_i}^2 \\
\tau =&\ \begin{bmatrix} l\ (f_1 + f_4 - f_2 - f_3) \\ l\ (f_1 + f_2 - f_3 - f_4) \\ \sum_{i=1}^{4} (-1)^{i+1}\ \tau_i \end{bmatrix} \\
x_{q\,att} =&\ \begin{bmatrix} \boldsymbol{q}^T & \omega^T \end{bmatrix}^T \\
\dot{x}_{q\,att} =&\ \begin{bmatrix} \frac{1}{2}\boldsymbol{q} \otimes \omega \\ J^{-1}(\tau - \omega \times J\,\omega) \end{bmatrix} = \begin{bmatrix} \frac{1}{2} S\,\omega \\ J^{-1}(\tau - \omega \times J\,\omega) \end{bmatrix}
\end{aligned}
\tag{3.13}
$$

where

- $x_{q\,att}$ is the state of the vehicle's attitude subsystem.
- τ_i denotes the torque of motor M_i.
- ω_{M_i} defines the motor's angular velocity.

- k_i represents the motor's thrust constant.
- f_i stands for the motor's thrust.
- L specifies the distance from the center of the quadcopter to the motors M_i. As we are assuming an X configuration, this distance is not the one used in the torques.
- τ indicates the torque that acts on the quadrotor frame from the motors.
- $J = \begin{bmatrix} J_{xx} & 0 & 0 \\ 0 & J_{yy} & 0 \\ 0 & 0 & J_{zz} \end{bmatrix}$ corresponds to the inertial matrix. As the quadcopter is almost symmetric with respect to all of its axis, it is safe to assume its diagonal.

Equilibrium Points. The equilibrium points are defined as the states and control inputs $\overline{x}, \overline{\tau}$ where $\dot{x} = \overline{0}$. Taking this into account, the equilibrium points of the attitude system (3.13) can be calculated when $\dot{x}_{q\,att} = \overline{0}$:

$$\begin{aligned} \dot{x}_{q\,att} &= \begin{bmatrix} \frac{1}{2}\boldsymbol{q} \otimes \omega \\ J^{-1}\left(\tau - \omega \times J\,\omega\right) \end{bmatrix} = \overline{0} \\ \Rightarrow \quad \overline{x}_{q\,att} &= \begin{bmatrix} \overline{\boldsymbol{q}} \\ \overline{0} \end{bmatrix}, \ \overline{\tau}_{q\,att} = \overline{0} \end{aligned} \tag{3.14}$$

This means that the attitude $\overline{\boldsymbol{q}}$ is constant in the absence of angular velocity and external torques.

Linearization. The model (3.13) can be used directly for simulation and application purposes. Nevertheless, it is difficult to design a control law because quaternions are not an intuitive representation.

A linear control using dual quaternions is proposed in Wang and Yu [2010], we will take this ideas to propose an algorithm using quaternions. The logarithmic mapping can be used to transform the quaternion part of the model into an axis-angle representation using equation (3.6). One advantage of this, is that the angular velocity used in the model (3.13) is also in an axis-angle representation, which means that $\dot{\bar{\theta}} = \omega$. This, along a cancellation of non-linear dynamics, leads to the following linear attitude model:

$$\begin{aligned} x_{att} &= \begin{bmatrix} \bar{\theta} & \omega \end{bmatrix} = \begin{bmatrix} \bar{\theta} & \dot{\bar{\theta}} \end{bmatrix} \\ \tau &:= J\, u_{att} + (\omega \times J\,\omega) \\ \dot{x}_{att} &= \begin{bmatrix} \dot{\bar{\theta}} \\ J^{-1}\,(\tau - \omega \times J\,\omega) \end{bmatrix} = \begin{bmatrix} \dot{\bar{\theta}} \\ u_{att} \end{bmatrix} \\ &= \begin{bmatrix} \bar{0} & I_3 \\ \bar{0} & \bar{0} \end{bmatrix} x_{att} + \begin{bmatrix} 0 \\ I_3 \end{bmatrix} u_{att} = A_{att}\, x_{att} + B_{att}\, u_{att} \end{aligned} \tag{3.15}$$

Notice that the pair A_{att} and B_{att} shows that the system (3.15) is controllable. The linear model (3.15) is the one that's going to be used to design the control law.

3.2.2 Translational Dynamics.

The translational system models the rest of the quadcopter dynamics. The state variable used is $x_{pos} = \begin{bmatrix} \bar{p}^T & \dot{\bar{p}}^T \end{bmatrix}^T$.

The drone's position is controlled using the total thrust vector. This vector's magnitude can be obtained by adding the thrust from all the motors $F_t = \sum_{i=1}^{4} f_i$. The total thrust's direction is fixed in the body frame, but is variable in the inertial frame and can be controlled if the attitude subsystem is completely controlled. This is the main relationship between both subsystems. The model, thus, can be described as:

$$\dot{x}_{pos} = \begin{bmatrix} \dot{\bar{p}} \\ \boldsymbol{q} \otimes \frac{F_t}{m} \otimes \boldsymbol{q}^* + \bar{g} \end{bmatrix} \tag{3.16}$$

where

- x_{pos} denotes the state of the vehicle's position subsystem.
- F_t defines the total thrust vector from the motors in the body frame.
- $\boldsymbol{q}$ represents the platform's attitude.
- m stands for the quadcopter's mass.
- $\overline{g}$ corresponds to the gravity's vector in the inertial frame.

3.3 Quadrotor Control Law.

The control presented in section 3.2.1, in the linear model (3.15), is an exact linearization under the assumptions that the inertial matrix J is known and there are no external torques acting on the quadcopter. In a practical sense, this doesn't hold true because the inertial matrix parameters are unknown and there may be external torques acting on the platform at any given time. In order to take into account these interactions, the following model will be used to obtain the control law:

$$\begin{aligned} & L_1, L_2 \in \mathbb{R}; \xi \in \mathbb{R}^3; \|\xi\| < L_2 + L_1 \|x\| \\ & \dot{x} = \begin{bmatrix} \bar{0} & I_3 \\ \bar{0} & \bar{0} \end{bmatrix} x + \begin{bmatrix} \bar{0} \\ I_3 \end{bmatrix} (u + \xi) = A\,x + B\,(u+\xi) \end{aligned} \tag{3.17}$$

where ξ are perturbations that include the external torques τ_{ext}, the term $-\omega \times J\,\omega$ and noise. Because of the nature of these terms, the vector ξ is assumed to be quasi-lipschitz, that is, $L_1, L_2 \in \mathbb{R}; \|\xi\| < L_2 + L_1 \|x\|$.

In order to stabilize these perturbations the following control law is proposed:

$$\begin{aligned} & \boldsymbol{q} \in \mathbb{H}; \overline{\theta} = 2 \ln \boldsymbol{q}, \omega, u \in \mathbb{R}^3; x = \left[\overline{\theta}^T \; \omega^T\right]^T \\ & K_1 > 0, K_2 > 0 \in \mathbb{R}^{3\times 6} \\ & u = -K_1\,x - K_2\,SIGN\,(x) \end{aligned} \tag{3.18}$$

Theorem 3.1. *The control law* (3.18) *stabilizes system* (3.17) *in the Lyapunov sense if there exists* K_1, K_2 *and* $P \in \mathbb{R}^{6\times 6}; P^T = P > 0$ *such that:*

$$\frac{\left|\left|-\left[(A - B\,K_1)^T\;P + P\;(A - B\,K_1)\right]\right|\right|}{2\,||P||\,||B||} > L_1$$

$$||K_2|| > L_2$$

Proof. Substituting (3.18) into (3.17) results in:

$$\dot{x} = (A - B\,K_1)\,x - B\;[K_2\,SIGN\,(x) + \xi]$$

The following Lyapunov candidate function is proposed:

$$\begin{aligned} &P \in \mathbb{R}^{6\times 6}; P > 0 \\ &V(x) = x^T\,P\,x \end{aligned} \tag{3.19}$$

The pair A and B is controllable because $rank\left\{\left[\begin{array}{cccccc} B & AB & A^2B & A^3B & A^4B & A^5B \end{array}\right]\right\} = 6$. This means that K_1 can be selected in a way that $(A - B\,K_1)^T\;P + P\;(A - B\,K_1) = -Q; Q \in \mathbb{R}^{6\times 6}; Q^T = Q > 0$. The derivative with respect of time of the proposed Lyapunov function is:

$$\begin{aligned} \dot{V}(x) =&\;\; 2\,x^T\,P\,\dot{x} \\ =&\;\; 2\,x^T\,P\;[(A - B\,K_1)\,x - B\,K_2\,SIGN\,(x) + B\,\xi] \\ =&\;\; -x^T\,Q\,x + 2\,x^T\,P\,B\,[-K_2\,SIGN\,(x) + \xi] \\ \leq&\;\; -||Q||\,||x||^2 + 2\,||P||\,||B||\,L_1\,||x||^2 - 2||P\,B||\,||K_2||\,||x|| + 2||P\,B||\,L_2\,||x|| \end{aligned}$$

The conditions $||Q|| > 2\,||P||\,||B||\,L_1$,$||K_2|| > L_2$ lead to $\dot{V}(x) < 0$, which means that the system is stable in the Lyapunov sense.

□

A good initial estimate of the matrix gain K_1 can be obtained using a linear quadratic regulator. The chattering around the origin because of the $SIGN$ function can be eliminated by substituting $SIGN(x)$ in (3.18) with an approximate function $TANH(x) = [\tanh x_1 \; \tanh x_2 \dots \tanh x_6]^T$.

Trajectory Tracking

The control presented in (3.18) stabilizes system (3.17) around the origin. In order for the quadcopter to follow a path, the desired variables $\overline{\theta}_d, \dot{\overline{\theta}}_d$ are created such that $x_d = \begin{bmatrix} \overline{\theta}_d^T & \dot{\overline{\theta}}_d^T \end{bmatrix}^T$. In the case when $\overline{\theta}_d$ is constant, $\dot{\overline{\theta}}_d = \overline{0}$. For any other case, these variables must be computed. With this, we have:

$$u = -K_1 \left(x - x_d\right) - K_2 \, SIGN \left(x - x_d\right) \tag{3.20}$$

This represents a change in origin, which the control law will try to follow. In order to stabilize a quadcopter in attitude and in the translational subsystems, this trajectory needs to guarantee the stability in the translational system (3.16).

3.4 Quadrotor Simulation.

The simulations were run using the Python language with the libraries NumPy Van Der Walt et al. [2011] SciPy Jones et al. [2001–], Control Systems Library Goppert et al. [2014–] and MatPlotLib Hunter [2007]. The program used can be found in appendix B

The simulated systems on Figures 3.3, 3.2 and 3.4 use the control law (3.20) in the quaternion system (3.13).

The used initial conditions were $x_0 = \begin{bmatrix} 0.70710678 \\ 0.09205746 \\ -0.64440223 \\ 0.27617239 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} \cos\frac{\pi}{4} \\ \begin{pmatrix} 0.13018891 \\ -0.91132238 \\ 0.39056673 \end{pmatrix} \sin(\frac{\pi}{4}) \\ \bar{0} \end{bmatrix}$.

The desired state was $x_d = \begin{bmatrix} 0.96592583 \\ 0 \\ 0 \\ 0.25881905 \\ \bar{0} \end{bmatrix} = \begin{bmatrix} \cos\frac{\pi}{12} \\ \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \sin(\frac{\pi}{12}) \\ \bar{0} \end{bmatrix}$

The used gain matrices were:

$$K_1 = \begin{bmatrix} 100 & 0 & 0 & 34.6410162 & 0 & 0 \\ 0 & 100 & 0 & 0 & 34.6410162 & 0 \\ 0 & 0 & 31.6227766 & 0 & 0 & 32.6074463 \end{bmatrix}$$
$$K_2 = \begin{bmatrix} 0.316227766 & 0 & 0 & 0.855836160 & 0 & 0 \\ 0 & 0.316227766 & 0 & 0 & 0.855836160 & 0 \\ 0 & 0 & 0.316227766 & 0 & 0 & 0.855836160 \end{bmatrix}$$

Volver a hacer las gráficas.

The state error of Figures 3.2, 3.3 and 3.4 can be seen on Figures 3.6, 3.5 and 3.7.

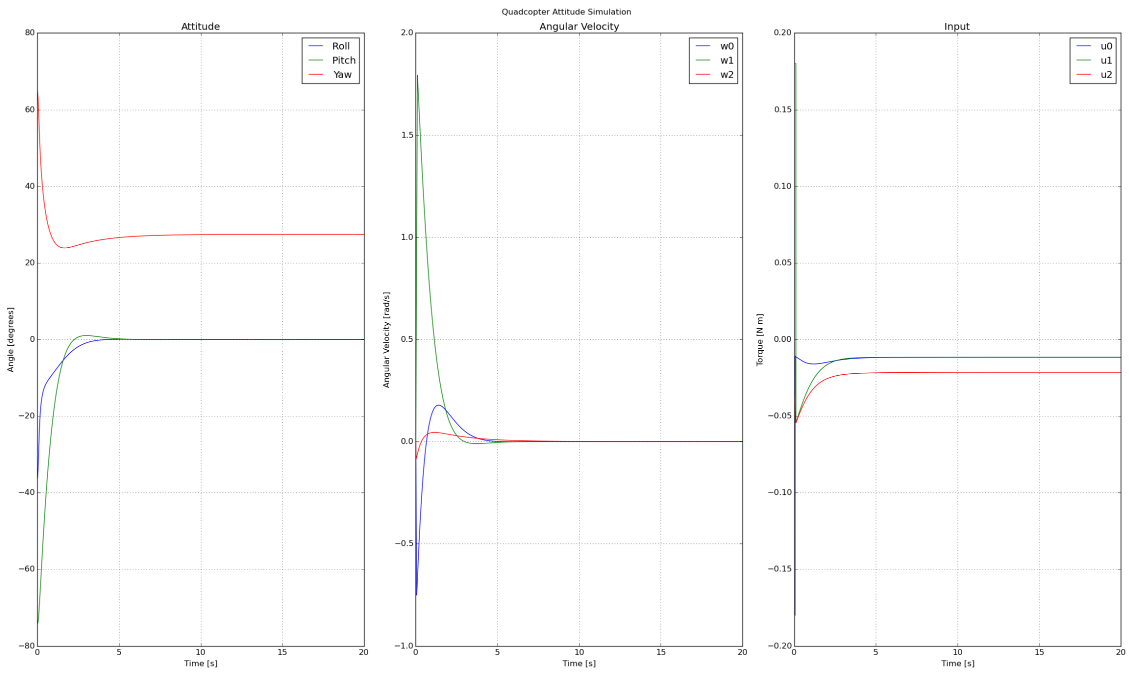

FIGURE 3.2: Quadcopter attitude simulation with constant perturbations.

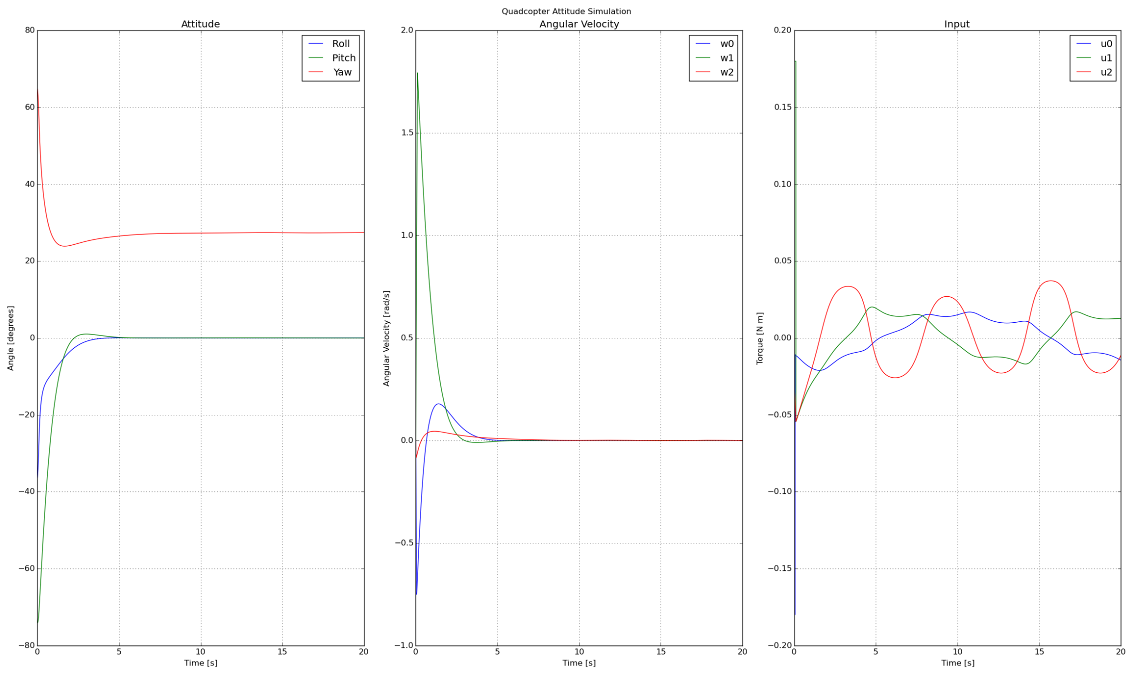

FIGURE 3.3: Quadcopter attitude simulation with sinusoidal perturbations.

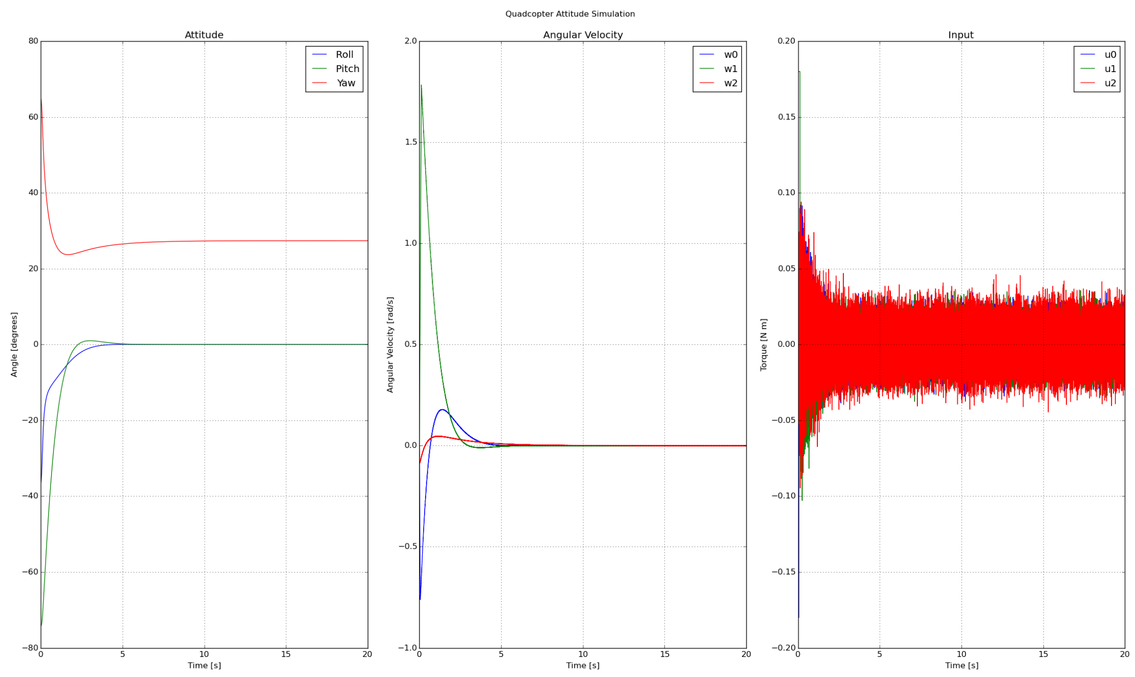

Figure 3.4: Quadcopter attitude simulation with random perturbations.

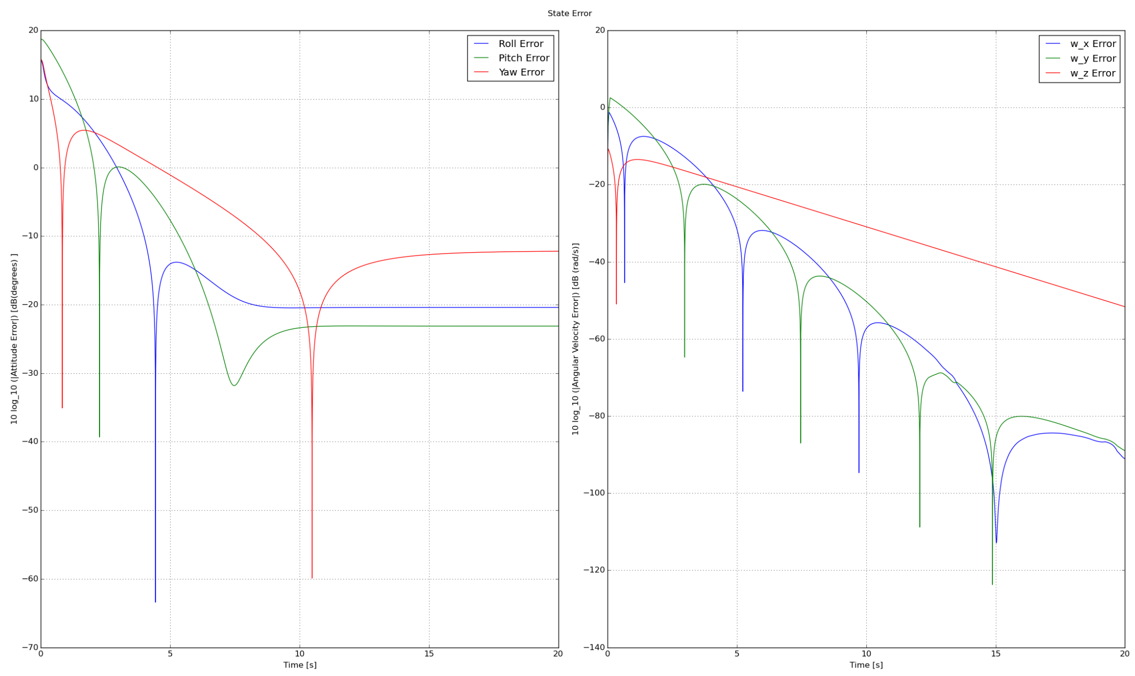

Figure 3.5: Quadcopter attitude errors simulation with constant perturbations.

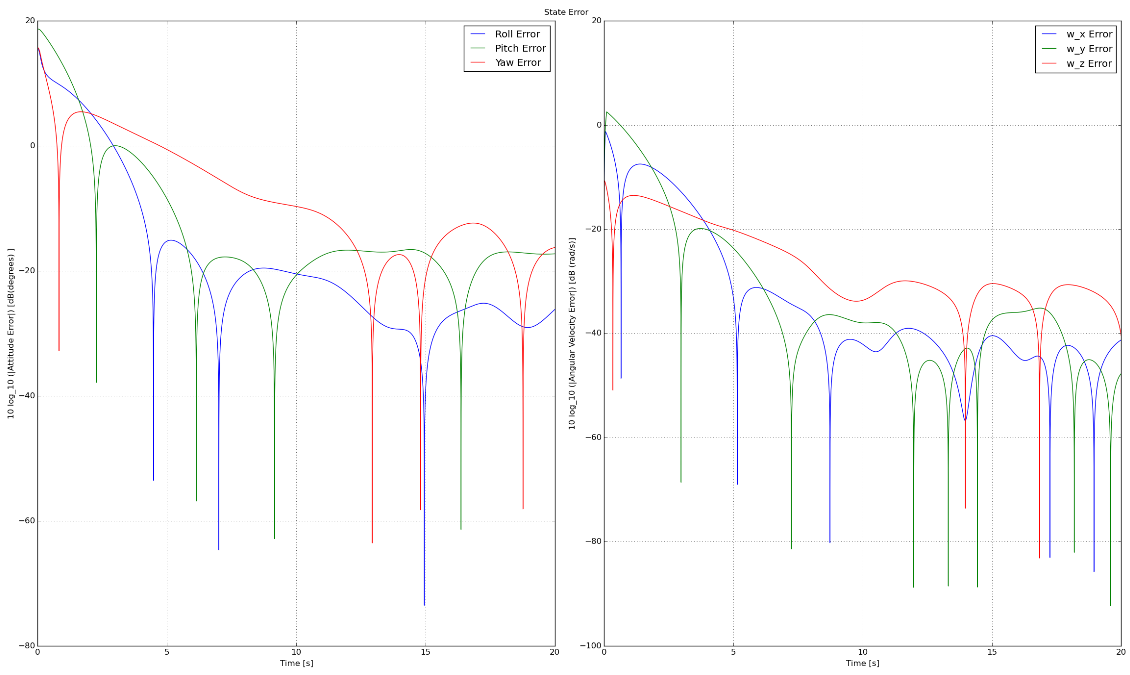

Figure 3.6: Quadcopter attitude error simulation with sinusoidal perturbations.

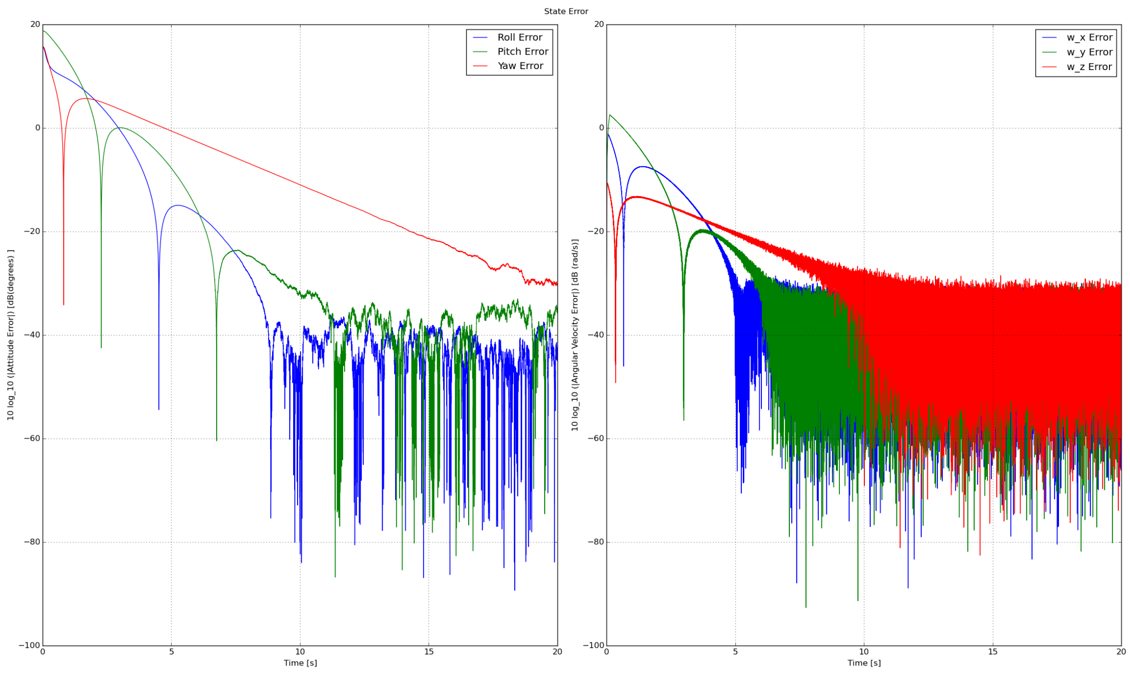

Figure 3.7: Quadcopter attitude error simulation with random perturbations.

Chapter 4

SLAM in 3D

In this chapter a quaternion observation model is used based on the model obtained in chapter 3. This model is used to construct a Kalman filter solution for the SLAM problem using a laser range-finder sensor, an IMU and an optical flow sensor. The RANSAC algorithm used to obtain the features and to match them is also presented.

4.1 Features

Features are the main elements used to describe a map in the state space representation. There are many kinds of features like lines, blobs, points, circles, edges, etc. The quality of the SLAM algorithm changes depending on the features used and some features are easier to recognize and use depending on the type of environment and the kind of sensor used. In order to have the best possible estimate, each feature should have the following characteristics:

- Robustness: It should be resistant to environment and sensor noise. Although the Kalman filter is based on the assumption of white noise, the system's error state covariance matrix determinant decreases more rapidly if the measurement variance is lower.

- Easy to identify: It should be easy to identify in a later measurement. The filter assumes that each feature is correctly paired thus an error in the matching process could result in an erroneous state prediction.
- Sufficiently large amount: There should be a sufficient amount of features to be able to have an estimate that reduces the system's error state covariance matrix determinant. If too few features are used, then this determinant could increase yielding an unstable solution. If there are too many, the system complexity increases and the computational requirements increase.
- Use small space: as each feature is stored in memory, it's more efficient if they use the least possible space. This isn't strictly necessary for the filter algorithm, but it should be noted when implementing it.

The objective of having a SLAM algorithm in a quadcopter platform is to have an accurate estimate of the vehicle's position, specially in GPS denied zones. Assuming that the most common environments where this might happen are urban corridors or indoor, it is possible to assume that one of the most common and reliable environment feature that will be encountered are straight walls perpendicular to the floor.

Unlike others sensors like stereoscopic cameras, laser range-finders produce very accurate scans of a plane. Planes give very limited information about 3D environments, so the position of the laser sensor on the drone determines what kind of information can be obtained. In this case, the laser plane is parallel to the xy body plane so that the sensor can give more information around the z-axis angle.

The reason for needing more information on the z-axis angle than on any other angle also comes from the environment. The yaw angle that the IMU located on the UAV gives relies heavily on the quality of the magnetometers measurements. Because the proximity of buildings and structures gives errors in the readings of these kind of sensors, the laser range-finder can compensate this noise by obtaining the yaw angle. The rest of the readings on the IMU can be considered accurate, so the angle in the x and y axes can be used to transform any wall encountered on the laser scans into a line parallel to the ground plane, assuming that the walls are perpendicular to the floor.

Feature Detection using RANSAC

There exists many techniques for extracting lines from a series of points like the Hough transform and the leasts squares line approximation. However, the least squares method takes into account all the points from the laser scan and not all those points can be considered as part of a line because more than one wall can be seen at the same time or, even worse, the points from the laser scan could belong to other objects that aren't walls. In the case of the Hough transform, it is robust enough to detect various lines but it usually uses an accumulator that tends to take a lot of space and it is more oriented towards image manipulations. Another vision algorithm that is robust enough to detect various lines and can be adjusted to not use many computational resources is the RANSAC algorithm.

RANSAC is an acronym for RANdom SAmple Consensus. It is a technique used to estimate in a robust way a data model using measurements contaminated with outliers. In the case of the quadrotor platform, it is used to extract line parameters from a laser scan despite the fact that some laser measurements may not correspond to the same wall or may not even come from any wall at all. The points that make up a line in the scan are called inliers, and the points that don't correspond to that line are referred to as outliers. The RANSAC algorithm tries to determine the inliers, and builds the model parameters using them. More information can be found on Zuliani [2009] and Civera et al. [2012].

The pseudo code of the RANSAC algorithm used can be found in Algorithm 1. When the laser scan measurements arrive, it is assumed that they contain various lines and outliers points.

As a line needs at least two parameters in order to be completely defined on a plane, two random points are selected in order to construct an hypothesis. The scanned points are evaluated in order to see the number of points that support the hypothesis. If the hypothesis has more points supporting it than the assumed best hypothesis, then the new hypothesis becomes the best hypothesis. This process is repeated for n_{hyp} times.

The number n_{hyp} can be obtained using the following equation:

$$n_{hyp} = \frac{\log(1-p)}{\log(1-(1-\varepsilon)^m)} \tag{4.1}$$

where p is the wanted probability of finding at least one line; ε denotes the outlier ratio with respect to the total scanned points; m represents the minimum number of points that are considered to be a valid line (or a wall in this case).

Once the best hypothesis is obtained from the scanned points, the model parameters are extracted using only the inliers from the best hypothesis with a leasts squares method. From this model, the other scanned points are evaluated and possible inliers are rescued. The final model parameters come from the clear inliers obtained from the best hypothesis and the rescued inliers using a least squares method. All the inliers for this model are extracted from the laser scan data until there aren't enough points to construct a new model or the best hypothesis didn't turn out to have enough points to initialize a new line model.

A single complex number can be used to store the parameters of each line, as they have an imaginary and a real part. In this work, the real part of the complex number represents the line's distance from the scan origin ρ_{F_i}, and the imaginary part represents the line's angle with respect to the scan origin θ_{F_i} yielding $F_i = \rho_{F_i} + \theta_{F_i}\,\hat{k}$. The imaginary part used is the $\hat{k}$ part of the quaternion imaginary vector. To convert from this representation to the polar complex form and vice versa, the following equations can be used:

$$polar\{F_i\} = \rho_{F_i}\, e^{\hat{k}\,\theta_{F_i}} \tag{4.2}$$

$$cartesian\{F_i\} = \rho_{F_i}\left(\cos\theta_{F_i} + \hat{k}\,\sin\theta_{F_i}\right) \tag{4.3}$$

Algorithm 1 RANSAC algorithm description.

```
$scan \leftarrow$ new scan ▷ new scan with model parameters and outlier points available
while $scan \rightarrow points \leq m$ do
    Create empty $bestInliers$
    for $n_{hyp}$ do
        $inliers \leftarrow$ Determine the number of inliers from a random minimun set.
        if $inliers \geq bestInliers$ then
            $bestInliers \leftarrow inliers$
        end if
    end for

    if $bestInliers \leq m$ then
        Break while loop            ▷ No new line was found on the remaining points
    end if

    $clearInliersModel \leftarrow$ Initialize model with the best inliers found.
    $rescuedInliers \leftarrow$ See which points become inliers using the $clearInliersModel$.
    $rescuedInliersModel \leftarrow$ Re-estimate the model with $bestInliers$ and $rescuedInliers$.
                                   ▷ Delete points that belong to the obtained model
    $scan \leftarrow [scan - (bestInliers + rescuedInliers)]$
end while                ▷ keep trying to find lines while there are still points left
```

The laser scan points are projected into the ground plane using data from the IMU. The objective of the SLAM algorithm, in this case, is to match and estimate the yaw orientation and position in the ground's $x - y$ plane. A laser scan from position $p \in \mathbb{C}$ and yaw orientation θ_z that hits the same wall as a stored feature $F_i = \rho_{F_i} + \theta_{F_i}\hat{k}$ has parameters $l_i = \rho_{F_i} - (Re\{p\} \cos\theta_{F_i} + Im\{p\} \sin\theta_{F_i}) + (\theta_{F_i} + \theta_z)\,\hat{k}$. This is a non-linear measurement function, hence the need to use an EKF if this kind of feature representation is used.

One thing to note is that this kind of method doesn't limit itself for line-based features. Any kind of model whose measurements can be contaminated by outliers can be used with this algorithm.

4.2 SLAM using Extended Kalman Filter

SLAM algorithms vary greatly depending on the kind of sensors used, the platform they are mounted into and the environmental conditions. Each implementation has to be tailored according to these factors. One of the standard methods used to solve SLAM is the extended Kalman filter (EKF) because almost all forms of maps and sensors can be adapted to it. All SLAM algorithms are compared to their EKF equivalent in order to test their efficiency and robustness, making the EKF the *de facto* SLAM solution. The filter guarantees a convergence to the real map and position if the model's linear approximation is accurate enough.

Kalman Filter

The Kalman filter was first described in Kalman et al. [1960], but a more detailed introduction can be found in Welch and Bishop [1995]. The filter is a combined predictor-corrector discrete estimator that minimizes the estimated error covariance for a linear stochastic system. The equations that make the filter can be separated in two different steps, the time update equations and the measurement update ones.

The filter assumes that the linear discrete system to be observed and the measurements are both contaminated by normal noise with zero mean.The equations of the process to be estimated are presented in the following equation:

$$\begin{aligned} x_k &= A\,x_{k-1} + B\,u_{k-1} + w_{k-1} \\ z_k &= H\,x_k + v_k \\ \\ & p(w) \sim N(0,Q); p(v) \sim N(0,R) \\ & x_k \in \mathbb{R}^n; u_k \in \mathbb{R}^m; z_k \in \mathbb{R}^l \\ & A \in \mathbb{R}^{n\times n}; B \in \mathbb{R}^{n\times m}; H \in \mathbb{R}^{l\times n} \end{aligned} \tag{4.4}$$

The objective of the filter is to obtain the best possible state estimates by comparing the system model to the actual process measurements and their parameters. In the time update part, an *a priori* state estimate $\hat{x}_k^-$ and error estimate covariance matrix P_k^- are obtained by propagating the previous corrected state ,$\hat{x}_{k-1}$ through the system's model equations. In the measurements update part of the algorithm, sensor data is used to correct the state estimate and obtain an *a posteriori* state estimate $\hat{x}_k$ and covariance error matrix P_k.

The time update equations are shown in equation (4.5), and the measurement update equations are shown in equation (4.6).

$$\begin{aligned} \hat{x}_k^- &= A\,\hat{x}_{k-1} + B\,u_k \\ P_k^- &= A\,P_{k-1}\,A^T + Q \end{aligned} \tag{4.5}$$

$$\begin{aligned} K_k &= P_k^-\,H^T\left(H\,P_k^-\,H^T + R\right)^{-1} \\ \hat{x}_k &= \hat{x}_k^- + K_k\left(z_k - H\,\hat{x}_k^-\right) \\ P_k &= (I_n - K_k\,H)\,P_k^- \end{aligned} \tag{4.6}$$

In the measurement update equations (4.6), the Kalman filter gain K_k determines the effect of $\hat{x}_k^-$ and z_k on $\hat{x}_k$.

Extended Kalman Filter

The Extended Kalman Filter (EKF) is a process estimation algorithm similar to the KF but that can be applied to non-linear processes and/or measurements. In this version, the estimation is linearized using a partial Taylor series expansion around the current estimate in order to determine better estimates from the non-linear process and measurements equations. The process to be estimated is described using the stochastic non-linear equation (4.7).

$$\begin{aligned} x_k &= f(x_{k-1}, u_{k-1}, w_{k-1}) \\ z_k &= h(x_k, v_k) \\ \\ p(w) &\sim N(0,Q); p(v) \sim N(0,R) \\ x_k &\in \mathbb{R}^n; u_k \in \mathbb{R}^m; z_k \in \mathbb{R}^l \end{aligned} \tag{4.7}$$

As with the KF, the EKF *a priori* state estimate is based on the model without the white noise and a linearized process noise variance is added to the *a priori* error estimate covariance matrix in the time update equations (4.8). A similar linearization is applied in the *a posteriori* measurements update equations (4.9).

$$\begin{aligned} \hat{x}_k^- &= f\left(\hat{x}_{k-1}, u_{k-1}, \overline{0}\right) \\ P_k^- &= A_k\, P_{k-1}\, A_k^T + W_k\, Q_{k-1}\, W_k^T \end{aligned} \tag{4.8}$$

where:

- $A_{[i,j]} = \dfrac{\partial f_{[i]}}{\partial x_{[j]}} \left(\hat{x}_{k-1}, u_{k-1}, \overline{0}\right)$ is the Jacobian of partial derivatives of $f(\cdot)$ with respect to x evaluated at the estimated point $\hat{x}_{k-1}$.
- $W_{[i,j]} = \dfrac{\partial f_{[i]}}{\partial w_{[j]}} \left(\hat{x}_{k-1}, u_{k-1}, \overline{0}\right)$ denotes the Jacobian of partial derivatives of $f(\cdot)$ with respect to w evaluated at the estimated point $\hat{x}_{k-1}$.

$$\begin{aligned} K_k &= P_k^-\, H_k^T \left(H_k\, P_k^-\, H_k^T + V_k\, R_k\, V_k^T\right)^{-1} \\ \hat{x}_k &= \hat{x}_k^- + K_k \left[z_k - h\left(\hat{x}_k^-, \overline{0}\right)\right] \\ P_k &= \left(I_n - K_k\, H_k\right) P_k^- \end{aligned} \tag{4.9}$$

where:

- $H_{[i,j]} = \dfrac{\partial h_{[i]}}{\partial x_{[j]}} \left(\hat{x}_k^-, \overline{0}\right)$ corresponds to the Jacobian of partial derivatives of $h(\cdot)$ with respect to x evaluated at the estimated point $\hat{x}_k^-$.
- $V_{[i,j]} = \dfrac{\partial h_{[i]}}{\partial v_{[j]}} \left(\hat{x}_k^-, \overline{0}\right)$ represents the Jacobian of partial derivatives of $h(\cdot)$ with respect to x evaluated at the estimated point $\hat{x}_k^-$.

The EKF in SLAM

SLAM can be seen as a stochastic problem in the sense that both the measurements and the stored feature parameters have noise. The filter state contains the pose of the vehicle and its velocity, along with all the relevant features needed to build a map. The odometry data obtained from sensors like the IMU and the optical flow sensor is propagated through the time update equations (4.8) and fused with the laser measurements in equations (4.9). As more measurements keep arriving, the estimate tends to improve the state error covariance.

Observation Model The discrete state space observation model is proposed as:

$$\begin{aligned}
&x_k \in \mathbb{C}^{4+N}; p_k, V_k \in \mathbb{C}; \theta_k, \omega_k \in \mathbb{I}; \rho_{F_i,k}, \theta_{F_i,k} \in \mathbb{R} \\
&x \in \mathbb{C}^{4+N}; x = \begin{bmatrix} p & V & \theta & \omega & F_1 & \cdots & F_N \end{bmatrix}^T \\
x_k =\ & f\left(x_{k-1}, w_{k-1}\right) \\
=\ & \begin{bmatrix} p_{k-1} + V_{k-1}\, e^{-\theta_{k-1}}\, \Delta t \\ V_{k-1} \\ \theta_{k-1} + \omega_{k-1}\Delta t \\ \omega_{k-1} \\ \rho_{F_1,\,k-1} + \theta_{F_1,\,k-1}\hat{k} \\ \vdots \\ \rho_{F_N,\,k-1} + \theta_{F_N,\,k-1}\hat{k} \end{bmatrix} + w_{k-1} \\
y_k =\ & h\left(x_k, v_k\right) \\
=\ & \begin{bmatrix} \rho_{F_1,\,k} - (Re\{p_k\}\cos\theta_{F_1,\,k} + Im\{p_k\}\sin\theta_{F_1,\,k}) + \theta_{F_1,\,k}\hat{k} + \theta_k \\ \vdots \\ \rho_{F_N,\,k} - (Re\{p_k\}\cos\theta_{F_N,\,k} + Im\{p_k\}\sin\theta_{F_N,\,k}) + \theta_{F_N,\,k}\hat{k} + \theta_k \end{bmatrix} + v_{k-1}
\end{aligned} \tag{4.10}$$

Where:

- p_k is the current position.

- V_k is the current velocity referenced in the body frame obtained from the optical flow sensor.
- θ_k is the current yaw angle.
- ω_k is the current yaw angular velocity obtained from the IMU.
- $\rho_{F_i,k}$ is the ith feature distance.
- $\theta_{F_i,k}$ is the ith feature angle.

The Jacobians used to build the EKF are:

$$A_k = \begin{bmatrix} 1 & e^{-\theta_{k-1}}\Delta t & -kV_{flow}e^{-\theta_{k-1}}\Delta t & 0 & \bar{0}_{1\times N} \\ 0 & 1 & 0 & 0 & \bar{0}_{1\times N} \\ 0 & 0 & 1 & \Delta t & \bar{0}_{1\times N} \\ 0 & 0 & 0 & 1 & \bar{0}_{1\times N} \\ \bar{0}_{N\times 1} & \bar{0}_{N\times 1} & \bar{0}_{N\times 1} & \bar{0}_{N\times 1} & I_N \end{bmatrix} \tag{4.11}$$

$$W_k = I_N \tag{4.12}$$

$$\frac{\partial h_i}{\partial F_i} = 1 + \frac{\left(Im\{p_k^-\}\cos\hat{\theta}_{F_i,k}^- - Re\{p_k^-\}\sin\hat{\theta}_{F_i,k}^-\right)\hat{k}}{2} \tag{4.13}$$

$$H_k = \begin{bmatrix} \frac{-e^{-\hat{k}\hat{\theta}_{F_1,k}^-}}{2} & 0 & \frac{1}{2} & 0 & \frac{\partial h_1}{\partial F_1} & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ \frac{-e^{-\hat{k}\hat{\theta}_{F_N,k}^-}}{2} & 0 & \frac{1}{2} & 0 & 0 & \cdots & \frac{\partial h_N}{\partial F_N} \end{bmatrix} \tag{4.14}$$

$$V_k = I_N \tag{4.15}$$

Feature matching The RANSAC algorithm described in 1 uses random points of the laser scan in order to search for possible inliers. An estimate of the form of the features in the laser scan can be obtained using the *a priori* state estimate of the EKF time update equations. This estimated features can be used to search for the matching features models without the need of random samples. If the model obtained doesn't have enough points, then it's assumed that the feature it's based on isn't visible in that moment in the laser scan. If enough points remain even after all the features have been searched for, then random points can be used to try to find new features in the scan.

The full SLAM algorithm is shown in Algorithm 2.

SLAM Simulation

Figures 4.1, 4.2, 4.3 and 4.4 show a simulation using The SLAM algorithm described in 2. As can be seen in Figures 4.1 and 4.2, the wall is relatively smooth and its shape can be seen.

In Figures 4.3 and 4.4, the trajectory of the quadcopter can be seen. The green line represents the real trayectory traversed by the quadcopter, and the blue line represents the SLAM estimation.

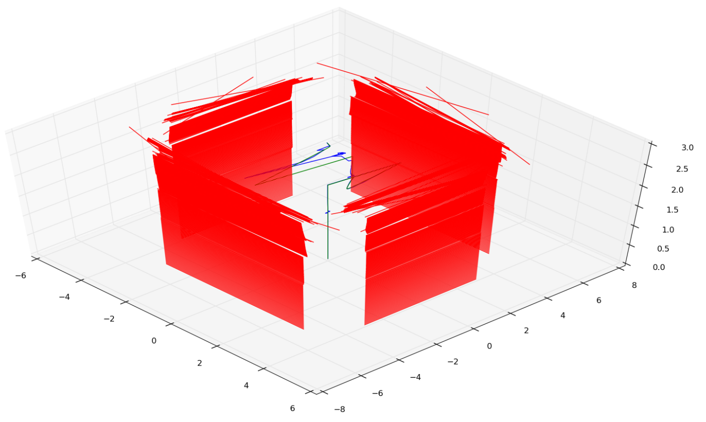

FIGURE 4.1: SLAM simulation with walls.

Algorithm 2 SLAM algorithm description.

```
$firstScan \leftarrow 1$    ▷ See if it's the first scan.
while scans are available do
    $scan \leftarrow$ new scan from the laser range-finder.
    $\boldsymbol{q}_{IMU} \leftarrow$ attitude from IMU.
    $\omega_{IMU} \leftarrow$ angular velocity from IMU.
    $V_{flow} \leftarrow$ translational velocity in the $xy$ plane from the optical flow sensor.
    $\Delta t \leftarrow$ time since last iteration.
    if firstScan then
        $groundProjection \leftarrow$ project the points in $scan$ to the ground plane using $\boldsymbol{q}_{IMU}$
        $F_0 \leftarrow$ extract the line features from $groundProjection$ using RANSAC.
        $\hat{x}_0 \leftarrow \begin{bmatrix} 0 & V_{flow} & (2\ln \boldsymbol{q}_{IMU})_z\,\hat{k} & (\omega_{IMU})_z\,\hat{k} & F_0 \end{bmatrix}^T$
        $\hat{\boldsymbol{q}}_0 \leftarrow \boldsymbol{q}_{IMU}$
        $firstScan \leftarrow 0$
    else
                                    ▷ EKF time update equations using odometry data.
        $\hat{x}_k^- = f\left(\hat{x}_{k-1}, V_{flow}, \omega_{IMU}, \overline{0}\right)$
        $A_k \leftarrow \hat{x}_{k-1}, \Delta t, V_{flow}$
        $P_k^- = A_k\, P_{k-1}\, A_k^T + Q$

                                    ▷ Feature extraction and matching.
        $groundProjection \leftarrow$ project the points in $scan$ to the ground plane using $\hat{\boldsymbol{q}}_k^-$
        $F_k \leftarrow$ extract the line features from $groundProjection$ using RANSAC and the estimated features locations $F_{k-1}$ to match the features discovered. It is assumed that features that were not matched weren't seen in the current scan.

                                    ▷ EKF measurement update equations using laser data.
        $H_k \leftarrow F_{k-1}, \hat{x}_k^-, F_k \rightarrow matchedFeatures$    ▷ Build $H_k$ using only matched features
        $K_k = P_k^-\, H_k^T \left(H_k\, P_k^-\, H_k^T + R_k\right)^{-1}$
        $z_k \leftarrow F_k \rightarrow matchedFeatures$    ▷ Build $z_k$ using only matched features
        $\hat{x}_k = \hat{x}_k^- + K_k\left[z_k - h\left(\hat{x}_k^-, \overline{0}\right)\right]$
        $P_k = (I_n - K_k\, H_k)\, P_k^-$
                                    ▷ Make sure that $\theta_k$ and $\omega_k$ are only imaginary.
        $\theta_k \leftarrow Im\{\theta_k\}\,\hat{k}$
        $\omega_k \leftarrow Im\{\omega_k\}\,\hat{k}$
                                    ▷ Calculate the current attitude.
        $\hat{\overline{\theta}}_k = \begin{bmatrix} (\theta_{IMU})_x \\ (\theta_{IMU})_y \\ Im\{\theta_k\} \end{bmatrix}$
        $\hat{\boldsymbol{q}}_k \leftarrow e^{\frac{\hat{\overline{\theta}}_k}{2}}$
                                    ▷ Add new features to the current state and the estimation error covariance matrix.
        $\hat{x}_k \leftarrow F_k \rightarrow newFeatures$
        $P_k \leftarrow F_k \rightarrow newFeatures$
    end if
end while
```

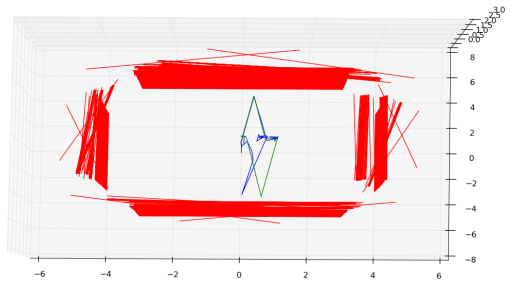

FIGURE 4.2: SLAM simulation with walls.

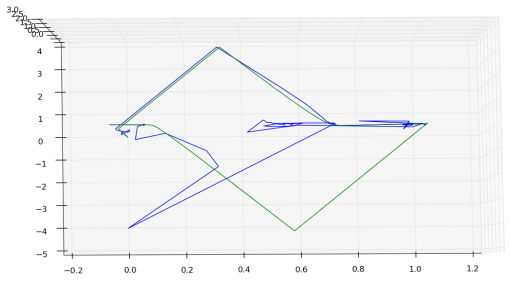

FIGURE 4.3: SLAM simulation without walls.

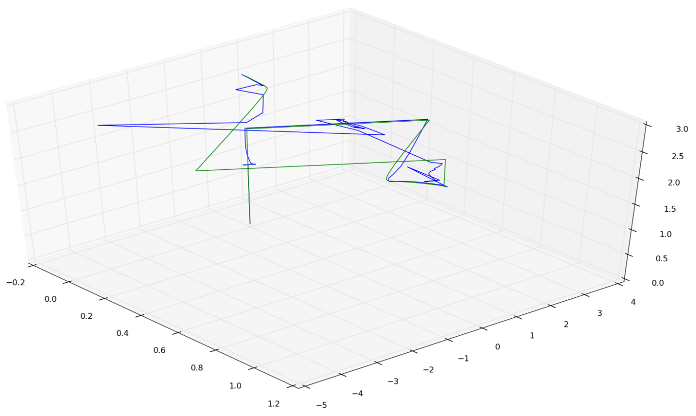

FIGURE 4.4: SLAM simulation without walls.

Chapter 5

Prototype

The design and the construction of a prototype suitable to test the desired SLAM algorithm and control law is presented in this chapter. The quadrotor is a platform that has seen a lot of use for surveillance and reckoning operations because it provides a very stable surface from where for cameras and other types of measurement instruments. The quadrotor was selected for this task mainly because it presents a very simple and reliable platform from which environment measurements can be obtained from a laser rangefinder, or any other SLAM-capable sensor for that matter. Also, the cost of production has decreased significantly, which means it's more likely to be used in a wide variety of situations and applications that could take advantage of a SLAM algorithm.

The general characteristics of the platform used as well as a list of its parts and their function towards generating an automated SLAM platform are presented below. As the focus of the SLAM research is mainly concentrated in the algorithm, the embedded system's description is presented and analyzed in more detail.

5.1 General Characteristics.

The platform was designed to be able to lift the weight of the laser rangefinder sensor, as well as an additional kilogram in order to expand the platform's capabilities for future research with additional sensors or instruments.

The quadrotor's parts are presented below.

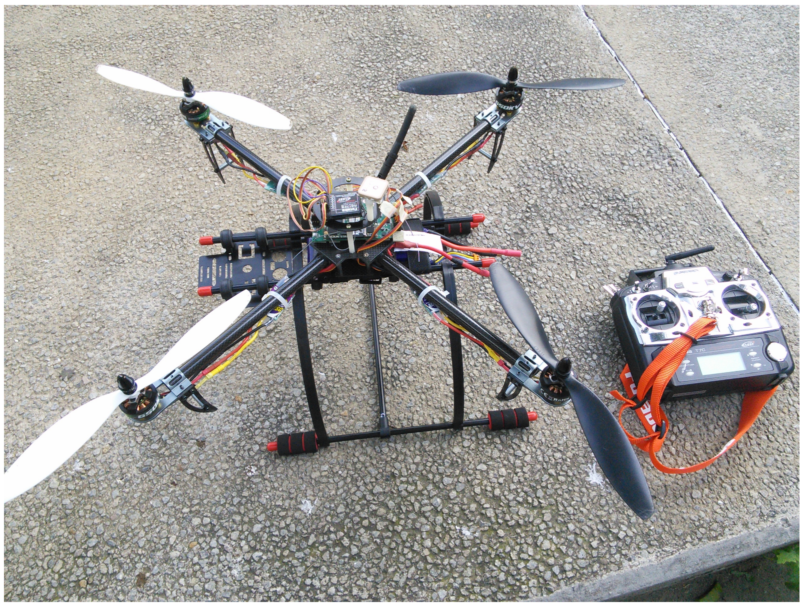

FIGURE 5.1: Quadcopter prototype used.

Components.

Frame. The frame consists of various carbon fiber and aluminum parts joined together with steel screws. It weights 280 g and has a total width and length of 550 mm.

Laser Rangefinder. The laser sensor used is a URG-04LX-UG01 from the Hokuyo brand. Its specifications are presented in Table 5.1.

Light source	Semiconductor laser diode (λ=785nm)
Laser safety	Class 1 (IEC60825-1)
Power source	5V DC $\pm$5% (USB buspower)
Current consumption	500mA or less (Rush current 800mA)
Detection distance	20mm $\sim$ 4000mm
Accuracy	Distance 20mm $\sim$ 1000mm $\pm$30mm Distance 20mm $\sim$ 4000mm $\pm$3% of measurement
Resolution	1 mm
Scan Angle	240°
Angular Resolution	0.36°
Scan Time	100msec /scan
Interface	USB Version 2.0 FS mode (12Mbps)
Preservation temperature	-25 $\sim$ 75°C
Ambient Light Resistance	10000Lx or less
Weight	Approx. 160 g
External dimension	(WxDxH) 50x50x70mm

TABLE 5.1: Characteristics of Hokuyo Laser Sensor.

Motors, propellers and Speed Controllers. The motors used are brushless outrunner motors. Their characteristics can be found in Table 5.2. The propellers used were of 11 (in) in length and 4.7 (in) of pitch inclination.

KV	650 RPM/V
LiPo cells	4s
Max Surge Watts	250 W
Working current	14 A
Max Current	17 A with 10s batteries
No Load Current	0.5 A
Intenal Resistance	0.178 Ω
Number of Poles	14
Dimensions	(Dia.xL) 35 x 25 mm
Shaft	4 mm
Weigth	58g

TABLE 5.2: Motor Specifications.

The speed controllers' specifications are listed in Table 5.3. These embedded systems' purpose is to drive the motor's speed to the desired setpoint in percentage.

Current Draw	30A Continuous
Voltage Range	2-4s Lipoly
BEC	0.5A Linear
Input Freq	1KHz
Weight	26.5g
Size	50 x 25 x 11mm

Table 5.3: Speed Controller's Specifications.

Power Source and Flight Duration. The batteries used where LiPo 4S1P (14.8 V) batteries with 3700 mAh of capacity. They have a constant discharge rate of 65C and weight 408 g. Their dimensions are: 138 x 44 x 31mm.

The duration of a constant flight (just hovering) with these batteries and without the laser rangefinder was of ten minutes.

External Compass. The external compass is used in order to avoid the noise from the motors' electromagnetic field.

Autopilot. The autopilot is an electronic card whose main function is to stabilize the quadcopter. It includes a variety of sensors for this task such as an accelerometer, a gyrometer, a magnetometer and a barometer. It has various communications protocols to receive the angular or translational set points, as well as waypoints to automatically follow a trajectory.

For safety reasons, its main set point source comes from the 2.4GHz radio receiver as this offers the most reliable communication connection and can override any other set point source.

The autopilot used is a commercial Gumstix AeroCore for DuoVero with GPS platform. Its specifications are listed on Table 5.4.

Embedded Computer. The embedded computer is used in conjunction with the autopilot. While the autopilot is used mainly for the control of the quadcopter, the

Product Family	DuoVero
Microcontroller	STM32F427 CortexTM-M4 180 MHz
GPS Module	UBloxTM NEO 7M GPS Receiver
Accelerometer	LSM303D 6-Axis Accelerometer/Magnetometer
Gyroscope	ST L3G4200D 3-Axis Gyroscope
Barometer	Measurement Specialties TMMS5611 Pressure Sensor
Headers	24-pin Header for 8 x PWM-controlled Motors 20-pin Header for STM32 GPIO/Breakout 20-pin Header for 5V Power 6 × 4-pin Headers for STM32 and DuoVero Breakout 6-pin Header for STM32 SPI Breakout
USB Ports	1 x USB On-The-Go
Serial Port	USB Console Port (USB-UART Bridge)
Dimensions	96.5mm x 50.1mm x 10.0mm

Table 5.4: AeroCore Autopilot Specifications.

embedded computer is used to run the SLAM algorithm without affecting the control law's functionality. The computer used is a DuoVero Zephyr Computer-On-Module. Its technical specifications are listed on Table 5.5.

5.2 Embedded System Description.

The embedded system presents two main processors. The microprocessor from the autopilot controls and assures the stability of the quadcopter. The embedded computer, on the other hands, is tasked with running the SLAM algorithm. A serial TTL connection between both systems is used to coordinate information between both systems.

The autopilot transmits attitude data to the embedded computer, which uses it, along the laser data, to perform the SLAM algorithm. The embedded computer, on the other part, can calculate and transmit the position estimates to the autopilot, which can be used to stabilize in position. Diagram 5.6 shows the different sensors and the connections with both processing boards.

Architecture	
Product Family	DuoVero
Central Processing Unit	Texas Instruments OMAP4430 1 GHz
Processor Architecture	ARM Cortex-A9
Digital Signal Processor	Texas Instruments C64x DSP
Memory	
RAM	1GB DDR SDRAM
Graphics	
Graphics Acceleration	PowerVR SGX540 with OpenGL
Connectivity	
Networking	802.11b/g/n WiFi
Bluetooth	3.0
Antennas	1 x U.Fl antenna connector
Storage	microSD Card Slot
Breakout	2 x 70-Pin Hirose DF40 Connectors
Audio Codec	Texas Instruments TWL6040
Power	
PMIC	Texas Instruments TWL6030
Power Input	2.5 – 4.8 V DC
Physical Specifications	
Dimensions	58mm x 17mm x 4.2mm
Weight	4.3g
Commercial Temperature Rating	0°C – 85°C
RoHS Compliant	Yes

Table 5.5: Duovero Embedded Computer Specifications.

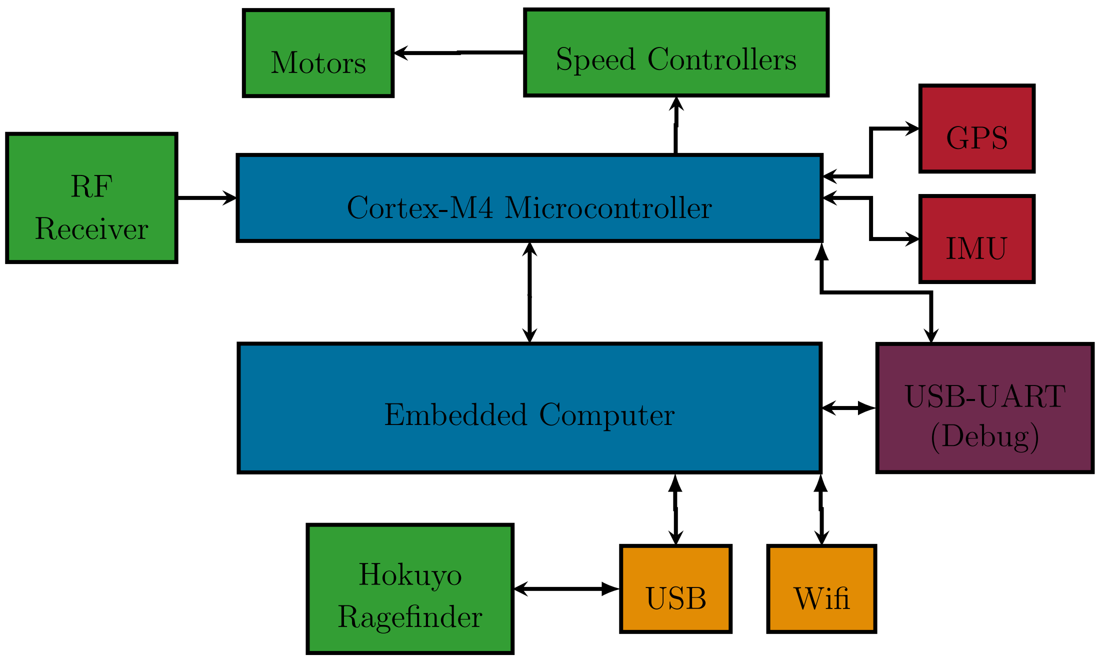

Table 5.6: Hardware Connection Diagram.

Chapter 6

Experimental Results

In this chapter the results of the experiments of the quadcopter's control law and of the SLAM algorithm are presented. The plataform used is described in chapter 5.

6.1 Attitude Stabilization

The attitude control law (3.20) was applied in the prototype presented in chapter 5. The parameters used in the test are shown in equation (6.1). The SIGN function was approximated with the TANH function.

$$\begin{aligned} K_1 &= \begin{bmatrix} 0.35 & 0 & 0 & 0.08 & 0 & 0 \\ 0 & 0.35 & 0 & 0 & 0.08 & 0 \\ 0 & 0 & 0.2 & 0 & 0 & 0.2 \end{bmatrix} \\ K_2 &= \begin{bmatrix} 0.01868069 & 0 & 0 & 0.04637922 & 0 & 0 \\ 0 & 0.01868069 & 0 & 0 & 0.04637922 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \end{aligned} \tag{6.1}$$

Figures 6.1 and 6.2 show the quadrotor attitude in quaternion and axis-angle representation.

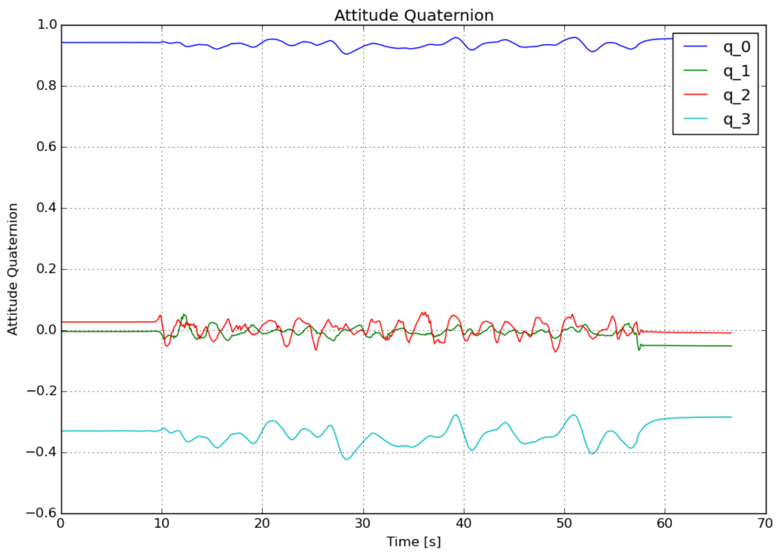

FIGURE 6.1: Attitude control response when using quaternion representation.

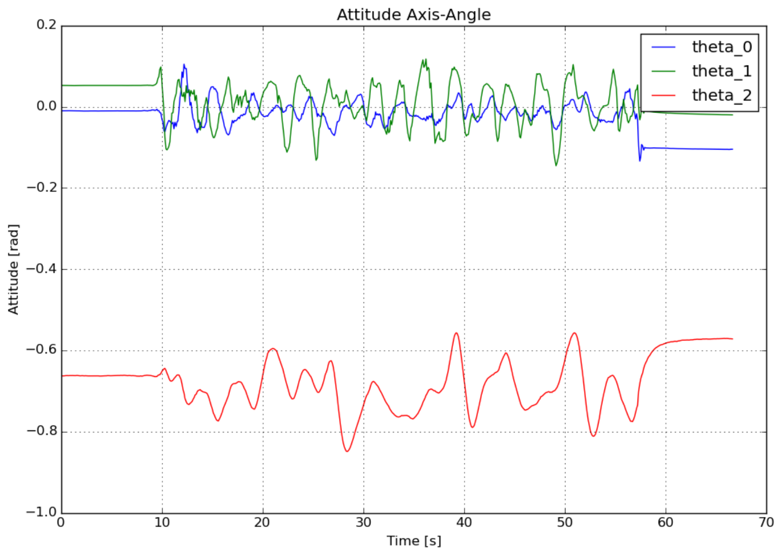

FIGURE 6.2: Attitude in axis-angle representation of attitude control test.

The reference values were given manually by a RF transmitter and were saved in the form of a desired quaternion at each time step. The generated desired trajectory, formed with the desired quaternion at each time step, can be seen on Figure 6.3 and in an axis-angle representation on Figure 6.4.

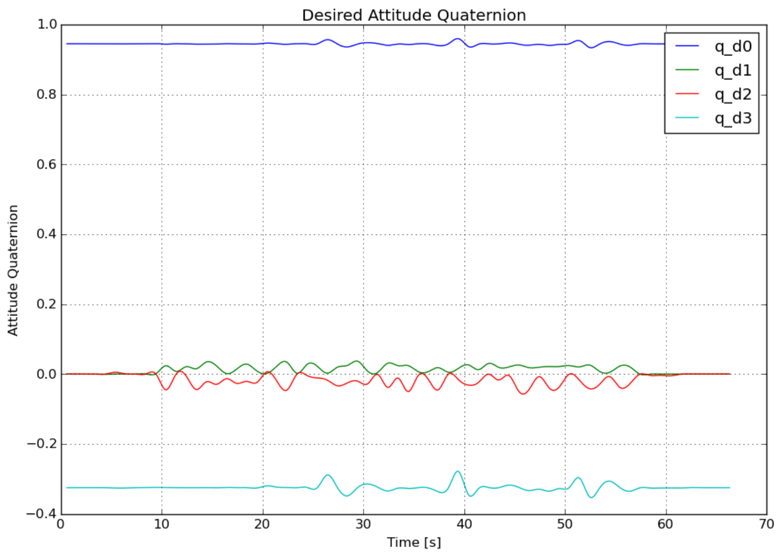

FIGURE 6.3: Desired attitude of attitude control test in quaternion representation.

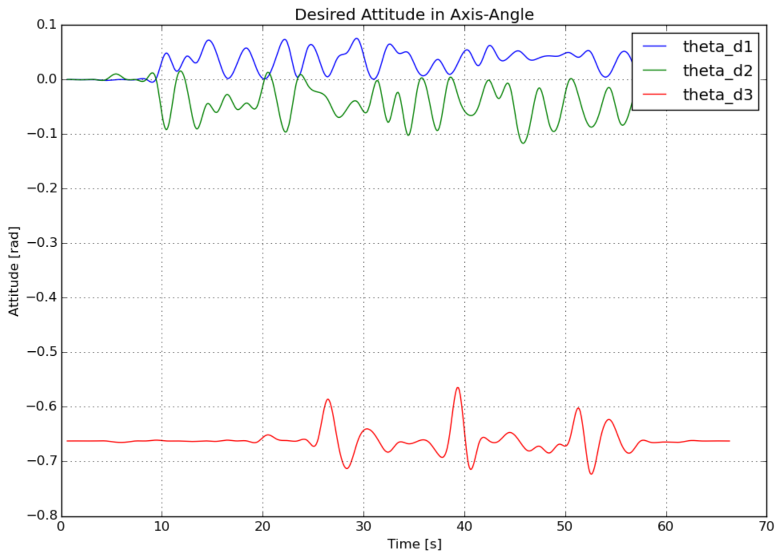

FIGURE 6.4: Desired attitude in axis-angle representation of attitude control test.

Figure 6.5 shows the quadrotor's angular velocity in $[rad/s]$

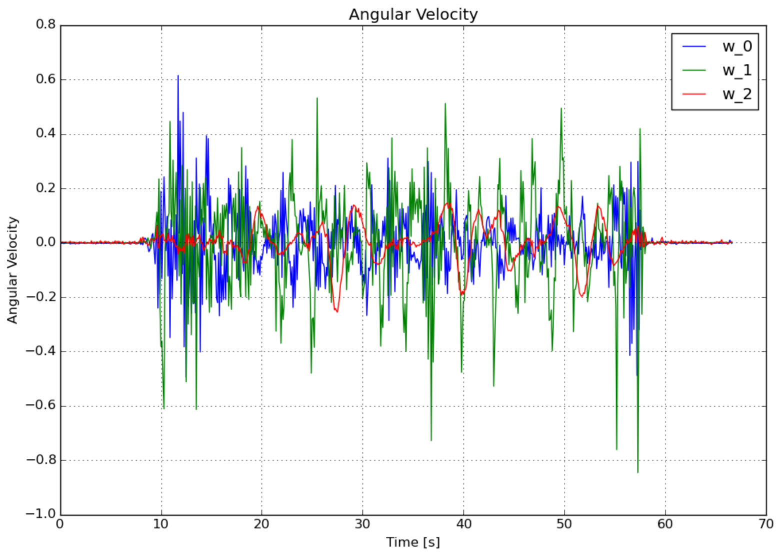

FIGURE 6.5: Angular velocity of attitude control test.

In Figure 6.6 the control inputs generated by the drone in order to follow the desired trajectory according to the attitude control law (3.20) can be seen.

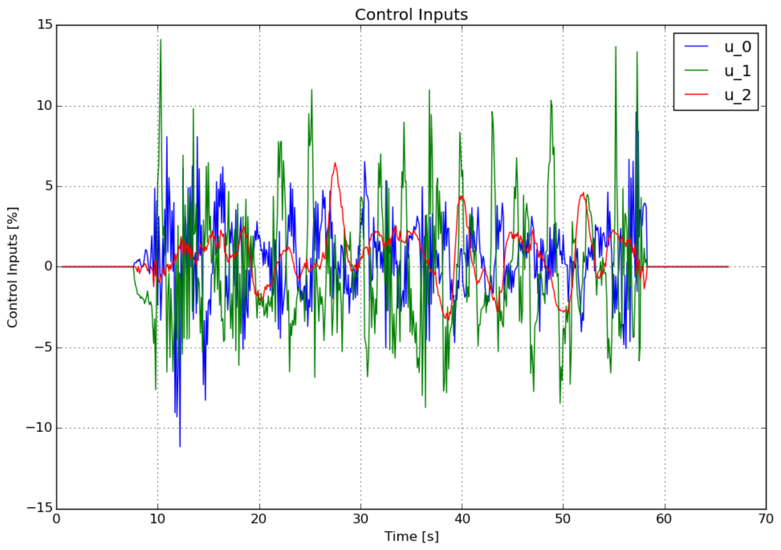

FIGURE 6.6: Attitude input torques in the body frame of attitude control test.

Figures 6.7, 6.8 and 6.9 display the attitude error in different representations. The error quaternion used in figure 6.7 is obtained from the desired quaternion $\boldsymbol{q}_d$ and the current quaternion $\boldsymbol{q}_{IMU}$ from the formula $\boldsymbol{q}_e \coloneqq \boldsymbol{q}_{IMU} \otimes \boldsymbol{q}_d^*$. This kind of error quaternion is a unit quaternion if both $\boldsymbol{q}_{IMU}$ and $\boldsymbol{q}_d$ are unti quaternions, and if $\boldsymbol{q}_{IMU} = \boldsymbol{q}_d$ the error quaternion becomes one.

The error in the axis-angle representation comes from the logarithm of the quaternion error, along with some algebraic manipulation thanks to the logarithmic properties, using the formula $\overline{\theta}_e = 2\,\log\left(\boldsymbol{q}_e\right) = 2\,\log\left(\boldsymbol{q}_{IMU} \otimes \boldsymbol{q}_d^*\right) = 2\,\log\left(\boldsymbol{q}_{IMU}\right) - 2\,\log\left(\boldsymbol{q}_d\right) = \overline{\theta}_{IMU} - \overline{\theta}_d$.

Figure 6.9 presents the axis-angle error using a logarithmic scale so that the smallest values in the errors can be visible. The scale used were decibels and the data used was the absolute value of the axis-angle error $10\,\log_{10}\left(|\overline{\theta}_e|\right)$.

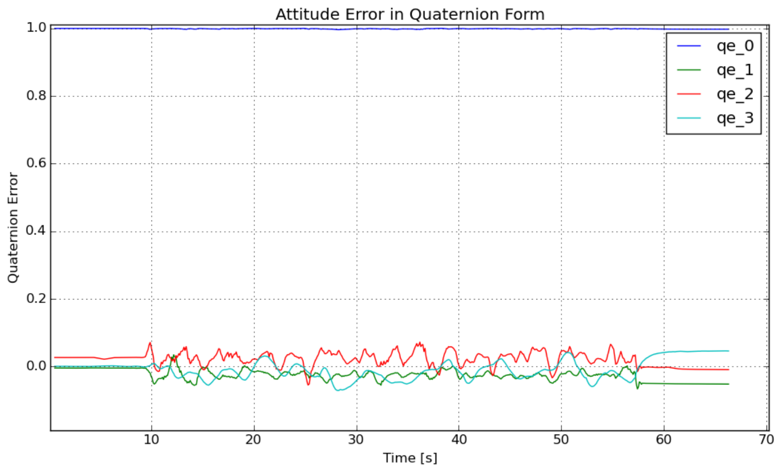

FIGURE 6.7: Attitude error in representation of attitude control test.

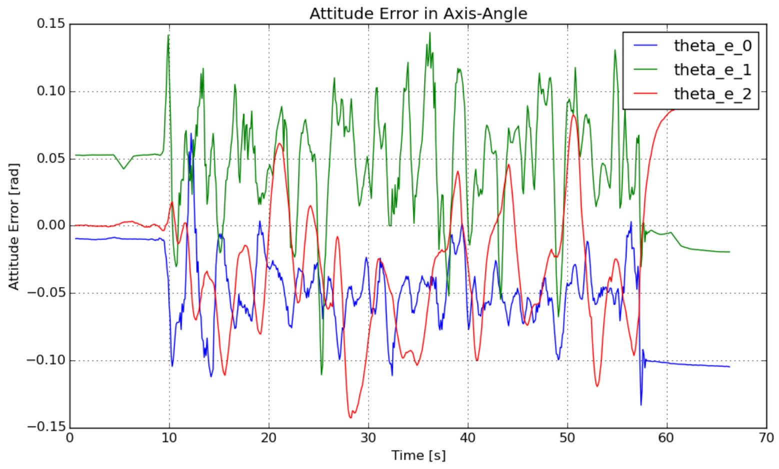

FIGURE 6.8: Attitude error in axis-angle representation of attitude control test.

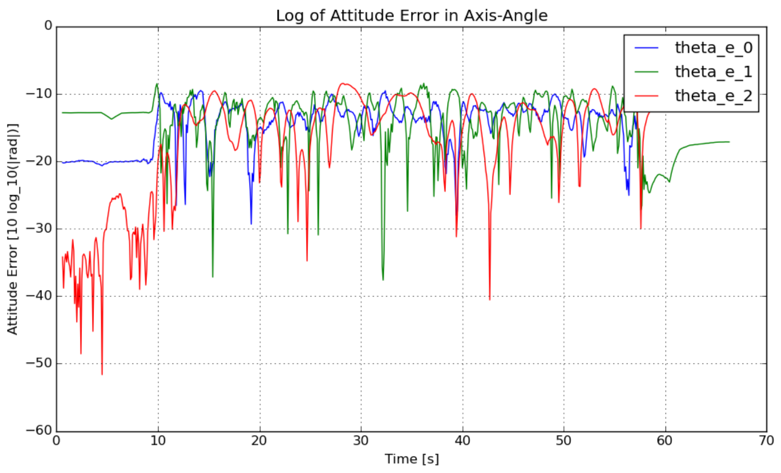

FIGURE 6.9: Attitude error in axis-angle decibels representation of attitude control test.

The constant lines at the beginning and at the end of the plots mark the takeoff and the landing of the platform on the ground.

The errors in Figures : 6.7, 6.8 and 6.9 corroborate that the system error was stable in the Lyapunov sense because it can be seen that It is bounded by a lower and upper interval. This is also true for the angular velocity, which is bounded in terms of the origin. It can be considered then that the quadrotor follows the commands given by the RF transmitter within a threshold, which is exactly the purpose of the attitude control law.

6.2 SLAM

The SLAM algorithm described in Algorithm 2 was implemented using ROS and the python programming language. The program used can be found on appendix C.

In this test, the platform flew near the wall of a building in order to be able to locate itself and generate a map of the surrounding are directly in front of it. Figures 6.10 and 6.11 display a point cloud representation of the generated map using *rviz*, which is a robot visualization tool for ROS. The different colors represent the different heights of the map points. The arrow represents the final position and orientation of the quadcopter on the map.

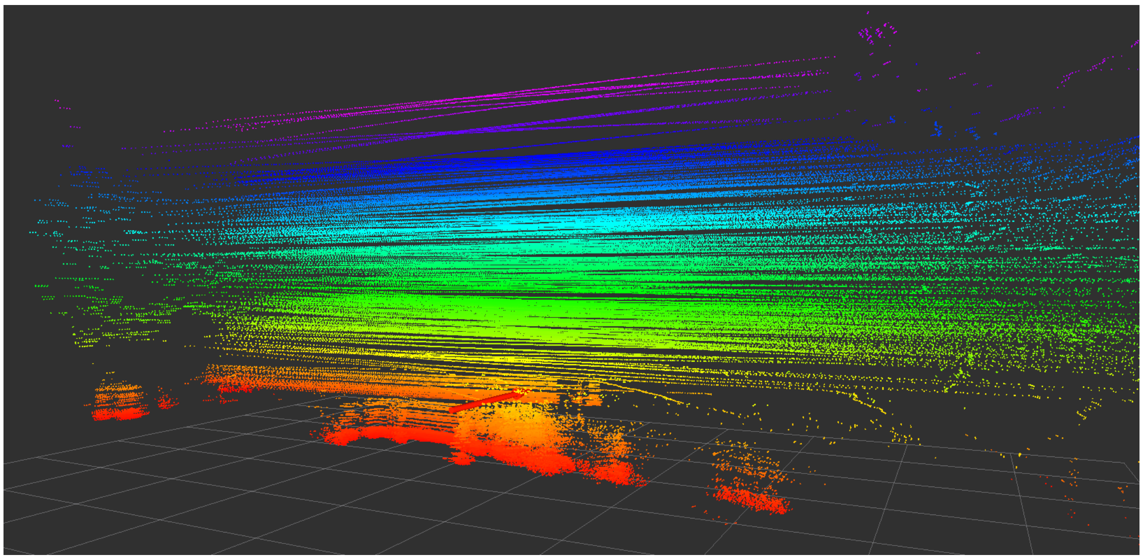

FIGURE 6.10: SLAM algorithm results map view 1

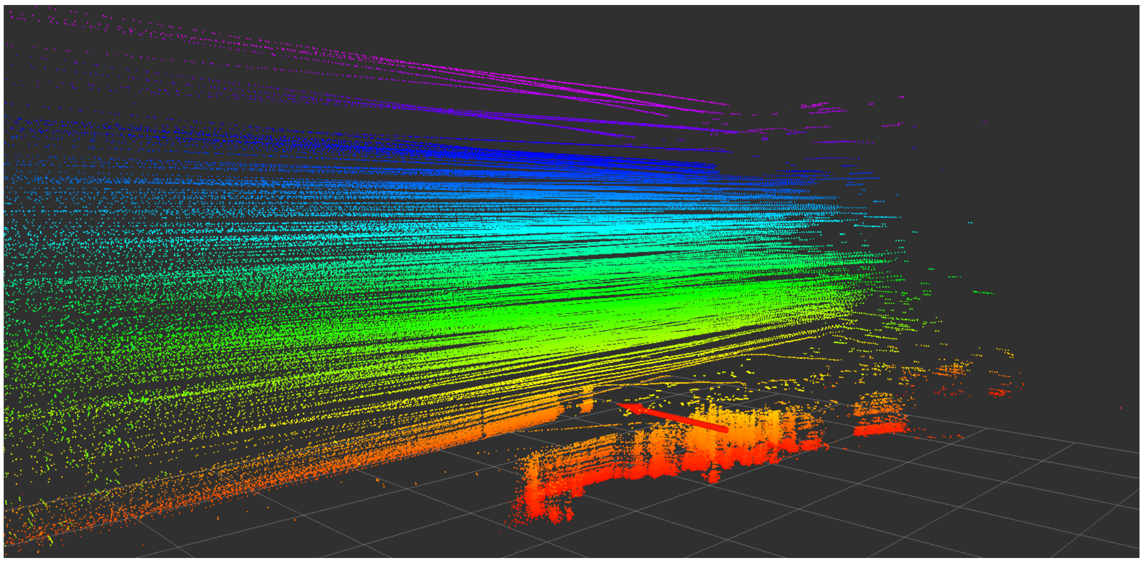

FIGURE 6.11: SLAM algorithm results map view 2

Figure 6.12 presents the position of the UAV with respect to the launch reference frame. The altitude is obtained from the barometer sensor. The x and y position are obtained using the SLAM algorithm together with the map points.

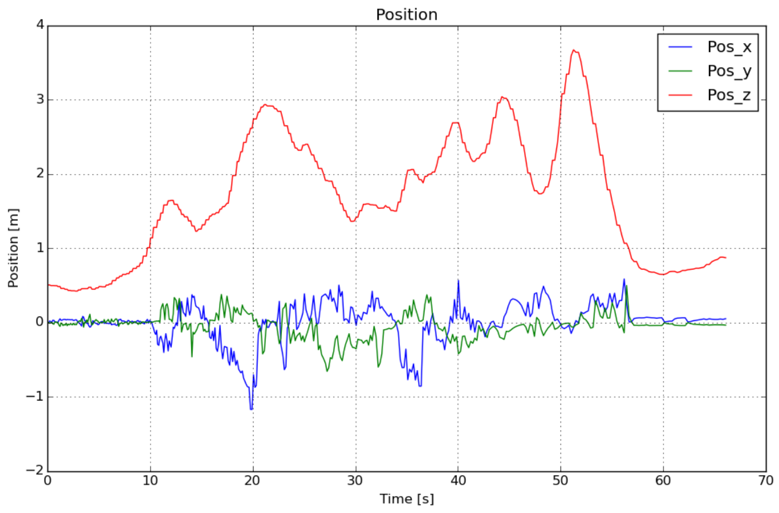

FIGURE 6.12: SLAM algorithm results position

Figures 6.13 and 6.14 represent the vehicle's attitude with respect to the local frame in quaternion and axis-angle form, respectively.

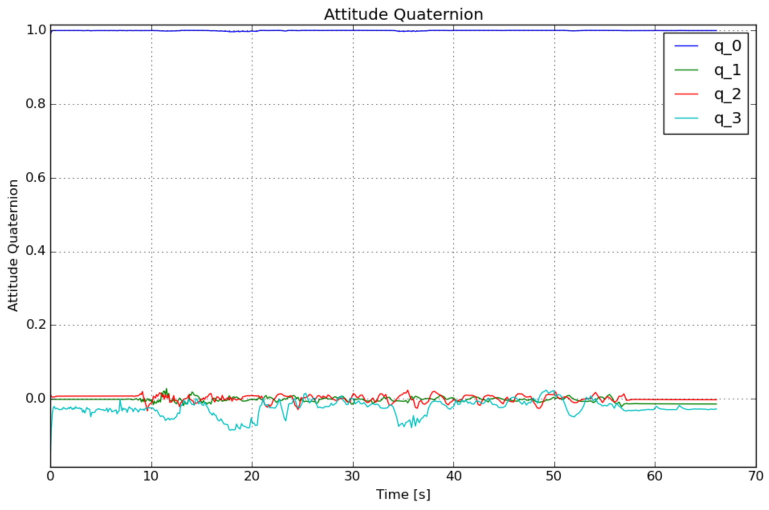

FIGURE 6.13: SLAM algorithm results quaternion attitude

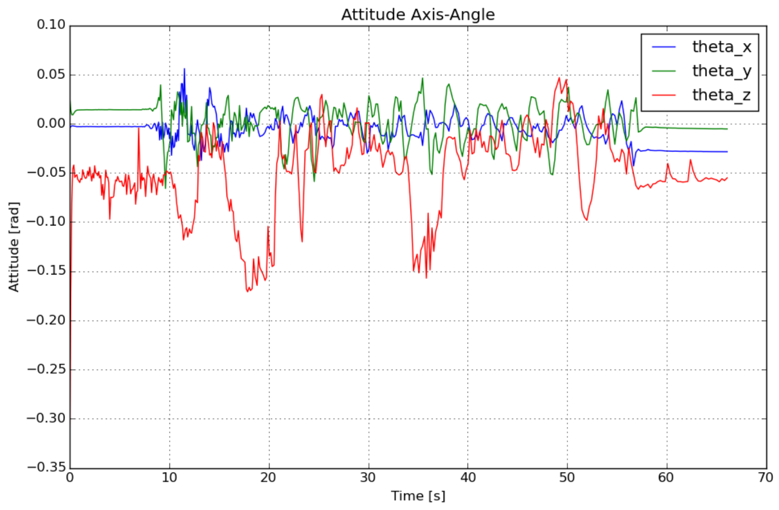

FIGURE 6.14: SLAM algorithm results axis-angle attitude

On Figures 6.10 and 6.11, the outline of the wall of the building can be seen clearly. There are other elements that can be seen in the figures, like some structures on the left and right sides of the wall, and part of the ground plants.

The attitude is the result of the fusion between the IMU and the SLAM data. It allowed to get a better measurement for the yaw angle in the absence of good magnetometer readings because of the proximity of the building.

Chapter 7

Conclusions

A linear system model using quaternions It was showed in chapter 3 that the attitude of a rigid body can be represented as a linear system without the use of any kind of approximation or linearization by using the axis-angle representation. It may be less intuitive to see the orientation of a vehicle in this way, but it is a more efficient way to use in terms of mathematical complexity.

Control laws are always tested against linear systems because they are the most studied type of dynamical systems in the sense of their controllability, observability and stability analysis. One of the most common practices is to linearize the non-linear system around an equilibrium point, thus using just a linear representation of a region of the system instead of the whole attitude state space. The system model presented in this work is linear in all the attitude space. The main drawback of using this kind of representation against a linearization using Euler angles, or even rotation matrices, is that it can increase computational complexity, requiring more operations to be done for the same result. To solve this, another kind of representation can be used, which involves the use of quaternions.

The relationship between the axis-angle representation and the quaternion one was also shown in chapter 3. The quaternion logarithmic mapping and exponential mapping give the basis to change from one type of model to another and vice-versa. The axis-angle model is the one that's used for analysis and for tests such as controllability, stability and observability. Then, using the quaternion logarithmic mapping and exponential mapping, this model can be changed to an analogous quaternion model. Quaternion's main operation is the product, which can be programmed as a sequence of products and sums, making it more efficient in terms of computational operations in comparison to the Euler representation, that needs to calculate sines and cosines. They use less space than rotations matrices, as only one quaternion is needed for the attitude in comparison to the nine data spaces needed for rotation matrices. In addition to all this, because quaternions used for rotations are unitary, they can be normalized to reduce numeric errors, making them more numerically stable than both Euler angles and rotation matrices.

The combination of these two representations allows a better use of the computational resources present on the UAV and a more elegant way to prove things in the control field. One thing to note, however, is that control laws applying the quaternion logarithmic mapping can have an abrupt discontinuity when generating quaternions from Euler angles. As quaternion rotation has the property that any rotation using quaternion $\boldsymbol{q}$ produces the same result as quaternion $-\boldsymbol{q}$, equation (3.8) only produces quaternions with scalar part greater or equal to zero. If only quaternions with positive scalar part are considered in the logarithmic mapping, a discontinuity arises when the quaternion scalar part approaches zero. In this case, the control tends to change signs abruptly, which can destabilize the system or even make the vehicle crash. To avoid this, the attitude data should be treated using the full quaternion space, or the discontinuity can be detected and corrected in an heuristic way.

A sliding modes control law The physical configuration of the prototype prevented the laser range finder to be mounted near the vehicle's center of mass because the landing gear would interfere with the laser readings. This meant that the sensor produced an external torque on the vehicle that needed to be compensated. This could be accomplished by using a counter weight on the quadcopter, but that would add mass, reduce flight time and doesn't compensate for any dynamic torques that may arise due to environmental conditions. The proposed solution was to make the control law robust enough to account for these external perturbations. Theorem 3.1 shows the sufficient conditions for the control law to achieve this objective using a sliding mode controller.

The control law can be seen as having two parts, a state feedback part and a signed part. The state feedback part allows the control to compensate perturbations that increase with the system state's norm. As the state variables tends to zero, the bounded perturbations' effect on the system increases. The signed part is added in order to reduce these effects on the system and to assure a finite convergence to the sliding surface, which is the origin in this case. In the practice, however, the sign function becomes problematic when applied directly to motors because it can reduce its working life.

There are many kinds of methods that deal with this shattering problem. Some of the more commonly used are: the use of a low pass filter to the output of the control law in order to reduce the shattering effect; the use of another function that is close enough to conserve some of the stability and convergence properties of the sign function, but different enough to dissipate the effect of the shattering on the system's actuators. In this work, the approximate function used was the hyperbolic tangent, because of its simplicity and because it doesn't require the addition of another dynamical system such as with the low-pass filter case would require. The implementation then becomes much simpler and robust.

It can be seen from the results obtained in and shown in Figure 6.8 that the system is stable in the Lyapunov sense as it was stated in Theorem 3.1. The attitude error and the angular velocity are bounded inside a region around the origin, even with the environment perturbations, the sensor weight and the changes in the desired trajectory from the RF signals references. This proves that the proposed control law is robust enough to handle the sensor weight and changes in the attitude reference, which makes it suitable for as a platform from which SLAM can be performed.

Yet another SLAM algorithm One of the most worrisome topics of UAVs is their safety, as they have the capacity of unintentionally harming people or objects with something as simple as falling onto them. As was the case with the early automobiles and planes, research focused on these platforms can increase their safety. In this case, SLAM research can have a direct impact on the how many persons could get hurt by using those kinds of platforms.

SLAM has many advantages for UAVs, which include the ability to navigate in unmapped environments, collision detection and to follow and plan trajectories. The application of SLAM to drones is an emerging field of study because recent advances in laser and imaging technology, as well as in the embedded circuits and computers, has made it possible to gather enough computational power in a sufficiently small form to be able to be used efficiently in UAVs. Yet, it is still a very resource-demanding task which could make drones safe to use in indoors environments and tight spaces safely. In order to solve the SLAM problem more efficiently, each SLAM algorithm implementation tends to be tailored to the specific platform and sensors that are used. The best configuration to each platform and sensors can only be obtained by trying different sensor-vehicles match-ups and comparing their results. That's one of the main reasons of trying different kinds of configurations.

Another way to solve the SLAM problem more efficiently is by changing the algorithm used. Almost all SLAM solutions are compared with the EKF-equivalent solution under the same configuration and model, making the EKF the main default with which to compare the effectiveness of any SLAM algorithm. In this work, instead of finding a more efficient way to solve the same problem, the SLAM model is changed and reconfigured to use complex and hyper-complex numbers (quaternions) in order to simplify it mathematically in order to reduce the computational complexity. It is shown with the results presented in Chapter 6 that with the new approach described in Chapter **??**, the SLAM problem can be solved using an EKF with less non-linear terms in the Jacobians compared to a pure EKF solution, making it more simple in mathematically and computationally. Because of the approach used, focused more in the model, this kind of technique can be used in combination to many other more efficient algorithms than the pure EKF in order to increase the effectiveness even more. Because of this, it can be safely stated that the objective of finding a more efficient method for solving the SLAM algorithm was accomplished.

What eyes to use? The prototype used in this work was assembled and built specifically for this work. The idea of using a laser range finder over other kinds of sensors was for precision in the measurements. However, because of the nature of range finder, it provides only data throughout a planed in 3D space, which has limited data with respect to the quadcopter's attitude and the environment because the sensor was fixed in the body frame. An imaging sensor such as a stereoscopic camera would have been able to provide more information about the surrounding environment, as well as for the vehicle's orientation. Even though laser sensors are more accurate, in this case the EKF filter would have been able to fuse various sensors to reduce this noise and still be able to have a wider range of information for the SLAM problem to be solved more efficiently.

The laser range finder could have been able to provide as much information as a camera array if it hadn't been fixed to the body frame, or several laser sensors would have been used. The change of orientation of the sensor would have enabled a better reconstruction of the 3D environment and a better estimation of the drone's attitude. Two or more laser sensors acquiring different planes with respect to the body frame would give enough information for the UAVs full attitude to be fully estimated in a more precise way.

Future Work.

Research in UAVs is still a very active development field of study. One of the main reasons is that drones represent one of the few platforms that can be used in almost any kind of environment, so many applications and systems that were once anchored to the ground because of the type of vehicle used or because of the cost of lifting them, are slowly starting to take off the earth.

A new revolution in robotics and autonomous systems is starting to brew, but it could quickly explode if not used carefully. Every day new models of multirotors and flying vehicles pop up on the world-wide market in a way that makes it almost impossible for legislation and people to keep up with in order to ensure a safe use of these vehicles.

The development of new SLAM algorithms and methods allows other kinds of research topics to emerge. Some of these are: path planning and following; obstacle avoidance and collision detection; object detection and manipulation and automatic take-off and landing. SLAM allows these platform to operate in unstructured environments, so any quadcopter application that starts without any prior knowledge of the surroundings or that doesn't have a good GPS signal can benefit from this implementation.

Appendix A

Quaternion Algebra

Quaternions are “hypercomplex” numbers, which means that they have three imaginary parts $\hat{\imath}, \hat{\jmath}, \hat{k}$ instead of one compared to complex numbers. They can be used to describe in a very simple mathematical and computational way rotations in three-dimensional space. When many methods use trigonometric functions, which are non-linear and suffer from numerical inaccuracy, quaternion rotations are simple in that they only need multiplications, divisions and sums to be implemented.

A.1 Notation.

In this work, over lined letters represent vectors in 3D space $\overline{(\cdot)} \in \mathbb{R}^3$. A quaternion is a four tuple that belongs to the $\mathbb{H}$ quaternion space. It can be seen as a number that contains one real part and three imaginary parts multiplied by their corresponding imaginary units $\hat{\imath}, \hat{\jmath}, \hat{k} \in \mathbb{I}$:

$$\begin{aligned} &q_0, q_1, q_2, q_3 \in \mathbb{R}; \boldsymbol{q} \in \mathbb{H}; \hat{\imath}, \hat{\jmath}, \hat{k} \in \mathbb{I} \\ &\boldsymbol{q} := q_0 + q_1 \, \hat{\imath} + q_2 \, \hat{\jmath} + q_3 \, \hat{k} \end{aligned} \tag{A.1}$$

As there are three different imaginary parts, these are often viewed as a vector in $\mathbb{R}^3$ space. Thus, $\mathbb{R}^3$ space can be seen as a subspace of $\mathbb{H}$ space and a $\mathbb{R}^3$ vector can be considered a pure imaginary quaternion.

$$\begin{aligned}&\overline{q} \in \mathbb{R}^3\\ &\boldsymbol{q} = q_0 + \begin{bmatrix} q_1 \\ q_2 \\ q_3 \end{bmatrix} = q_0 + \overline{q}\end{aligned} \tag{A.2}$$

A.2 Quaternion Operations.

Because of its significance, historically as well as in the definition of the quaternion space itself, the main operation of quaternions is the multiplication. Other operations and properties arise from this definition, like the conjugate and the norm.

Product. The quaternion product between quaternions $\boldsymbol{q}, \boldsymbol{p} \in \mathbb{H}$, expressed as a sum between a scalar real part and an imaginary vector $\boldsymbol{q} = q_0 + \overline{q}; \boldsymbol{p} = p_0 + \overline{p}$, is defined in the following manner:

$$\boldsymbol{q} \otimes \boldsymbol{p} := (q_0 + p_0 - \overline{q} \cdot \overline{p}) + (q_0\,\overline{p} + p_0\,\overline{q} + \overline{q} \times \overline{p}) \tag{A.3}$$

Some properties can be seen from this definition. One of the most important is that quaternion product is not commutative. Which means that $\boldsymbol{q} \otimes \boldsymbol{p} \neq \boldsymbol{p} \otimes \boldsymbol{q}$. This is because of the same non-commutativity property of the cross product used on the definition.

Sum. The sum of quaternions $\boldsymbol{q}$ and $\boldsymbol{p}$ is simply defined as the sum of each of its elements, like this:

$$\begin{aligned} &\boldsymbol{q}, \boldsymbol{r} \in \mathbb{H} \\ &\boldsymbol{q} + \boldsymbol{r} := q_0 + r_0 + \overline{q} + \overline{r} \end{aligned} \tag{A.4}$$

The set of all quaternions with operations addition and multiplication defines a non-commutative division ring. See Kuipers [1999] for more information on this matter.

Conjugate. The conjugate of quaternion $\boldsymbol{q}$ is denoted like:

$$\boldsymbol{q}^* := q_0 - \overline{q} \tag{A.5}$$

The conjugate of a product of quaternions is:

$$(\boldsymbol{q} \otimes \boldsymbol{r})^* = \boldsymbol{r}^* \otimes \boldsymbol{q}^* \tag{A.6}$$

This can be proven by expanding the corresponding products.

Norm. The norm of a quaternion is defined by:

$$||\boldsymbol{q}||^2 := \boldsymbol{q} \otimes \boldsymbol{q}^* = q_0^2 + q_1^2 + q_2^2 + q_3^2 \tag{A.7}$$

Inverse. The quaternion product forms a closed-loop group, that is, the product of two non-null quaternions is another quaternion. This means that for any non-null quaternion there exists an inverse quaternion such that:

$$\begin{aligned} \boldsymbol{q}^{-1} &:= \frac{\boldsymbol{q}^*}{||\boldsymbol{q}||} \\ \boldsymbol{q} \otimes \boldsymbol{q}^{-1} &= \boldsymbol{q}^{-1} \otimes \boldsymbol{q} = 1 \end{aligned} \tag{A.8}$$

Appendix B

Simulations' Code

Attitude Control Simulations

The following code was used to generate Figures 3.3 and 3.2.

```
# -*- coding: utf-8 -*-

import numpy as np
from numpy.linalg import norm
from numpy import concatenate as cat, array, cross
from control.matlab import lqr
from scipy.integrate import odeint
from pylab import *
from quat_library import *

#Final simulation time and resolution
tf = 2.5e1
res = tf*1e3

#Quadrotor parameters
m = 1.3
g = 9.81
vg = array([0,0,-g], dtype=float)
sp = 2*(1.3-4*0.065)*0.06**2/5
```

```
armxy = 4*0.065*0.2**2
armz = 4*0.065*0.28**2
J = np.diag([sp + armxy, sp + armxy, sp + armz])
th = g*m*1.5
tau_max = 0.6*0.3


#Model parameters
A = np.zeros((6,6))
A[:3,3:6] = np.eye(3)
B = np.zeros((6,3))
B[3:6,:] = np.eye(3)

#Get gain matrices
K_1, P, E = lqr(A,B,diag([1e1,1e1,1e0,1,1,1])*1e3,diag([1,1,1])*1e0)
K_2, P_2, E = lqr(A,B,diag([1,1,1,1,1,1])*1e-1,diag([1,1,1])*1e0)

#Noise parameters
Q = -( (A-B.dot(K_1)).transpose().dot(P) + P.dot((A-B.dot(K_1))) )
L1 = norm(Q)/(2*norm(P)*norm(B))
L2 = norm(K_2)

#Initial conditions
U = []
th_0 = np.deg2rad(90)
e_0 = array([1,-7,3], dtype=float)
e_0 /= norm(e_0)
q_b = e_0*np.sin(th_0/2)
x0 = array([np.cos(th_0/2),q_b[0],q_b[1],q_b[2],0,0,0],dtype=float)
x0[:4] /= norm(x0[:4])
if x0[0] < 0:
    x0[:4] *= -1

#Desired attitude
yd = np.deg2rad(30)
qyd = array([np.cos(yd), 0,0, np.sin(yd)])
qyd /= norm(qyd)
```

```

def f(x, t):
    global xi
    x = array(x,dtype=float)
    q = x[:4]
    q /= norm(q)
    w = x[4:7]
    #Calculate the state error
    x_err = cat((log_q( mult_q( q,conj_q(qyd) ) ), w))
    #Calculate the control
    tau = -K_1.dot(x_err) - K_2.dot(np.tanh(x_err))
 #A physical contrain
    tau[np.abs(tau) > tau_max] = np.sign(tau[np.abs(tau) >
    tau_max])*tau_max

    #Add noise
#     xi = np.cos([0.3*t, 0.5*t, t]) #Sinusoidal noise
    xi = ones((3,)) #Constant noise
    xi/= norm(xi)
    xi *= (L1 + L2*norm(x_err))/(1+1e-10)
#     xi = np.random.normal(size=(3,),scale = (L1 +
    L2*norm(x_err))/(1+1e-10)) #Random noise

    dx = cat((0.5*mult_q(q,np.concatenate(([0],w
    ))),np.linalg.inv(J).dot(tau - np.cross(w,J.dot(w)) ) + xi  ))

    return dx

#Finde the solution
t = np.linspace(0, tf, res)
y = odeint(f, x0, t)

#Solution using simple euler recursion, in case the derivate is not
     continuous because of the noise.
#y = []
#dt = tf/res
#for T in t:
```

```
#       if T == 0:
#           y_ant = x0
#       else:
#           y_ant = f(y_ant, T)*dt+y_ant
#       y_ant[:4] /= norm(y_ant[:4])
#       y.append(y_ant)
#y = array(y)


count = 0
#Calculate Euler angles for a more intuitive representation
q = y[:,:4]
euler = cat((    np.arctan2(q[:,0]*q[:,1] + q[:,2]*q[:,3],
    1-2*(q[:,1]**2 + q[:,2]**2) )[None,:],
                 np.arcsin(2*(q[:,0]*q[:,2] - q[:,3]*q[:,1]))[None,:],
                 np.arctan2(2*(q[:,0]*q[:,3] + q[:,1]*q[:,2]),
    1-2*(q[:,2]**2 + q[:,3]**3))[None,:] ), axis = 0)

#Plot
subplot(1,3,1)
title(u"Attitude")
xlabel(u"Time [s]")
ylabel(u"Angle [degrees]")
if abs(np.rad2deg(euler.max())) < 0.1 and abs(np.rad2deg(euler.min()))
    < 0.1 :
        ylim([-0.1,0.1])
plot(t, np.rad2deg(euler[0,:]), label=u"roll")
plot(t, np.rad2deg(euler[1,:]), label=u"pitch")
plot(t, np.rad2deg(euler[2,:]), label=u"yaw")
grid()
legend()

suptitle(u"Quadcopter Attitude Simulation")

subplot(1,3,2)
title(u"Angular Velocity")
xlabel(u"Time [s]")
```

```
ylabel(u"Angular Velocity [rad/s]")
count = 0
for x in y.transpose()[4:7]:
    if abs(x.max()) < 0.1 and abs(x.min()) < 0.1:
        ylim([-0.1,0.1])
    plot(t, x, label=u"w" + str(count))
    count += 1
grid()
legend()

#Recalculate the control
u = []
for x in y:
    q = x[:4]
    q /= norm(q)
    w = x[4:7]
    x_err = cat((log_q( mult_q( q,conj_q(qyd) ) ), w))
    u.append(-K_1.dot(x_err) - K_2.dot(np.tanh(x_err)))
u = np.array(u)

u[np.abs(u) > tau_max] = np.sign(u[np.abs(u) > tau_max])*tau_max

subplot(1,3,3)
title(u"Input")
xlabel(u"Time [s]")
ylabel(u"Torque [N m]")
count = 0
for x in u.transpose():
    if abs(x.max()) < 0.1 and abs(x.min()) < 0.1:
        ylim([-0.1,0.1])
    plot(t, x, label=u"u" + str(count))
    count += 1
grid()
legend()

#Plot the state error
figure()
```

```
suptitle(u"State Error")

subplot(1,2,1)
e_ref = array([np.arctan2(qyd[0]*qyd[1] + qyd[2]*qyd[3],
    1-2*(qyd[1]**2 + qyd[2]**2)),
                  np.arcsin(2*(qyd[0]*qyd[2] - qyd[3]*qyd[1])),
                  np.arctan2(2*(qyd[0]*qyd[3] + qyd[1]*qyd[2]),
    1-2*(qyd[2]**2 + qyd[3]**3))])
error = cat(((euler[0,:] - e_ref[0])[None,:],  (euler[1,:] -
    e_ref[1])[None,:],  (euler[2,:] - e_ref[2])[None,:] ), axis=0)
plot(t, 10*np.log10(np.abs(np.rad2deg(error.transpose()[:,0]) )),
    label=u"Roll Error" )
plot(t, 10*np.log10(np.abs(np.rad2deg(error.transpose()[:,1]) )),
    label=u"Pitch Error" )
plot(t, 10*np.log10(np.abs(np.rad2deg(error.transpose()[:,2]) )),
    label=u"Yaw Error" )
xlabel("Time [s]")
ylabel(u"10 log_10 (|Attitude Error|) [dB(degrees) ]")
grid()
legend()

subplot(1,2,2)
plot(t, 10*np.log10(np.abs(y[:,4] )), label=u"w_x Error" )
plot(t, 10*np.log10(np.abs(y[:,5] )), label=u"w_y Error" )
plot(t, 10*np.log10(np.abs(y[:,6] )), label=u"w_z Error" )
xlabel("Time [s]")
ylabel(u"10 log_10 (|Angular Velocity Error|) [dB (rad/s)]")
grid()
legend()

ion()
show()
```

Appendix C

Code for the Embedded Computer

Attitude Control Simulations

The following code was used to generate Figure 6.10. This file is used to produce the ROS node used:

```
#!/usr/bin/env python
import rospy
import sensor_msgs.point_cloud2 as pc2
from rospy.numpy_msg import numpy_msg
from sensor_msgs.msg import Imu, LaserScan, PointCloud2
from mavros_extras.msg import OpticalFlowRad
from geometry_msgs.msg import PoseStamped
import numpy as np
from SLAM import slam as slam_ob

def quatRot(q, v): #q v q^*
    q, v = np.array(q,dtype=float), np.array(v, dtype=float)
    q /= np.linalg.norm(q)
    if len(v.shape) == 1:
        v = np.concatenate(([0],v))
    else:
        v = np.concatenate((np.zeros((1,v.shape[1])), v ),axis=0)
    r = np.concatenate((-q[1:4].dot(v[1:4,:])[None,:], q[0]*v[1:4,:] + np.cross(q[1:4], v[1:4,:], axis = 0)),axis=0)
    return q[0]*r[1:4,:] - q[1:4,None].dot(r[None,0,:]) - np.cross(r[1:4,:],q[1:4],axis=0)

def quatMult(q,r):
    q,r = np.array(q,dtype=float), np.array(r,dtype=float)
    return np.concatenate(([q[0]*r[0]-q[1:4].dot(r[1:4])], q[1:4]*r[0] + q[0]*r[1:4] + np.cross(q[1:4],r[1:4])))

def quat2mat(q):
    q /= np.linalg.norm(q)
    r = np.empty((3,3))
    r[0,:] = np.array([1 - 2*q[2]**2 - 2*q[3]**2, 2*(q[1]*q[2] - q[3]*q[0]), 2*(q[1]*q[3] + q[2]*q[0]) ], dtype=float)
    r[1,:] = np.array([2*(q[1]*q[2] + q[3]*q[0]), 1 - 2*q[1]**2 - 2*q[3]**2, 2*(q[2]*q[3] + q[1]*q[0]) ], dtype=float)
    r[2,:] = np.array([2*(q[1]*q[3] - q[2]*q[0]), 2*(q[2]*q[3] + q[1]*q[0]), 1 - 2*q[1]**2 - 2*q[2]**2 ], dtype=float)
    return r

def Lrot(li):
    return np.array([[    0., -li[0], -li[1], -li[2]],
                     [ li[0],     0.,  li[2], -li[1]],
                     [ li[1], -li[2],     0.,  li[0]],
                     [ li[2],  li[1], -li[0],     0.]],dtype=float)

def M_Prot(pi):
    return np.array([[    0., -pi[0], -pi[1], -pi[2]],
                     [ pi[0],     0., -pi[2],  pi[1]],
                     [ pi[1],  pi[2],     0., -pi[0]],
                     [ pi[2], -pi[1],  pi[0],     0.]],dtype=float)



def attUpdate(data):
    global q_imu, w_imu
    q_imu = np.array([data.orientation.w,
                              data.orientation.x,
                              data.orientation.y,
                              data.orientation.z])
    q_imu /= np.linalg.norm(q_imu)
    w_imu = np.array([data.angular_velocity.x,
                              data.angular_velocity.y,
                              data.angular_velocity.z],dtype=float)
#       rospy.loginfo(q_imu)

scanWasUpdated = False

def scanUpdate(data):
    global ranges, angles, scanWasUpdated
    ranges = data.ranges
    angles = np.linspace(data.angle_min, data.angle_max,
        ranges.shape[0])
    not_valid = np.isnan(ranges) + np.isinf(ranges)
    ranges = ranges[np.logical_not(not_valid)]
    angles = angles[np.logical_not(not_valid)]
    scanWasUpdated = True
#       rospy.loginfo(ranges)

velWasUpdated = False

def velUpdate(data):
    global velWasUpdated, vel, velQuality
    vel = np.array([data.flow_comp_m_x, data.flow_comp_m_y])
    velQuality = data.quality
    velWasUpdated = True

altitude = None
altWasUpdated = False

def altUpdate(data):
    global altitude, altWasUpdated
    altitude = data.pose.position.z
    altWasUpdated = True
```

```
#     rospy.loginfo(altitude)

def slam():
    global scanWasUpdated, ranges, angles, q_imu, altitude,
   velWasUpdated, vel, altWasUpdated, w_imu
#     Initialize
    rospy.init_node('slam', anonymous=True)
    resolution = rospy.get_param("resolution", 0.01)
    max_dist = rospy.get_param("max_dist", 0.5)
    frame_id = rospy.get_param("inertial_frame_id", "local_origin")

#     Subscribe to topics
    rospy.Subscriber("/mavros/imu/data", numpy_msg(Imu), attUpdate)
    rospy.Subscriber("/scan", numpy_msg(LaserScan), scanUpdate);
    rospy.Subscriber("/mavros/optical_flow",
   numpy_msg(OpticalFlowRad), velUpdate);
    rospy.Subscriber("/mavros/position/local", numpy_msg(PoseStamped),
   altUpdate);

    pub = rospy.Publisher('map', PointCloud2, queue_size=5)
    pose = rospy.Publisher('pose', PoseStamped, queue_size=5)

    rate = rospy.Rate(100) # 100hz
    x_odo = np.zeros((6,))
    slam_scan = slam_ob()
    seqNum = 0
    q_hat = np.array([1.0,0,0,0])

    while not rospy.is_shutdown():
        #Sliding Mode Observer for position estimate
        if velWasUpdated:
            x_odo[:2] += x_odo[3:5]*0.1 + 0.1*(vel - x_odo[3:5])
            x_odo[3:5] += 0.45825757*(vel - x_odo[3:5])
            velWasUpdated = False
        else:
            x_odo[:2] += x_odo[3:5]*0.1
            x_odo[3:5] *= 0.5
```

```
#           rospy.loginfo("x_odo: " + str(x_odo))
        if altWasUpdated:
            x_odo[2] = altitude
#             x_odo[5] += 0.45*np.tanh(altitude - x_odo[2])
            altWasUpdated = False

        #Do something if there's a scan
        if scanWasUpdated and altitude != None and q_imu != None:
            #check if it's a valid scan
            if ranges != None and ranges.shape[0] != 0 and
    ranges.shape == angles.shape:
                odoData = {'altitude': altitude, 'velocity':
    x_odo[3:5], 'q_imu': q_imu, 'w': w_imu}
                scanData = {'ranges': ranges, 'angles':angles}
                slam_scan.updateStates(scanData, odoData)
                x_odo[:3] = slam_scan.pose['position'].copy()
                q_hat = slam_scan.pose['attitude']

            #Publish Map topic
            Map = PointCloud2()
            Map = pc2.create_cloud_xyz32(Map.header, slam_scan.mapPnts)
            Map.header.frame_id = frame_id
            pub.publish(Map)
            #Publish pose topic
            pose_hat = PoseStamped()

            pose_hat.header.frame_id = frame_id
            pose_hat.header.stamp = rospy.Time.now()
            pose_hat.header.seq =  seqNum
            seqNum += 1

            q_show = quatMult([1,0,0,0], q_hat)

            pose_hat.pose.orientation.x = q_show[1]
            pose_hat.pose.orientation.y = q_show[2]
            pose_hat.pose.orientation.z = q_show[3]
            pose_hat.pose.orientation.w = q_show[0]
```

```

                pose_hat.pose.position.x = slam_scan.pose['position'][0]
                pose_hat.pose.position.y = slam_scan.pose['position'][1]
                pose_hat.pose.position.z = slam_scan.pose['position'][2]

                pose.publish(pose_hat)

                scanWasUpdated = False

#           Map = pc2.create_cloud_xyz32(Map.header,
    [[0,1,2],[3,4,5],[6,7,8]])
#           Map.header.frame_id = frame_id
#           pub.publish(Map)

#           rospy.loginfo(Map)
            rate.sleep()

if __name__ == '__main__':
    try:
        slam()
    except rospy.ROSInterruptException:
        pass
```

The following code is used as the SLAM object that was called in the previous code:

```
# -*- coding: utf-8 -*-

import numpy as np
from numpy import array
from numpy.linalg import norm
from scipy.optimize import leastsq

#Define functions for quaternion operations

def log_q(q):
    q = array(q, dtype=float)
    v_norm = norm(q[1:4])
    if v_norm == 0:
        return array([0,0,0], dtype=float)
    else:
        return np.arccos(q[0])*q[1:4]/v_norm

def exp_q(v):
    v = array(v, dtype=float)
    v_norm = norm(v)
    if v_norm == 0:
        return array([1,0,0,0], dtype=float)
    else:
        return np.concatenate(([np.cos(v_norm)],v*np.sin(v_norm)/v_norm))

def mult_q(q, r):
    q, r = array(q, dtype=float), array(r, dtype=float)
    if len(q.shape) == 1 and len(r.shape) == 1:
        return np.concatenate(([q[0]*r[0]-q[1:4].dot(r[1:4])], q[0]*r[1:4] + q[1:4]*r[0] + np.cross(q[1:4], r[1:4])))
    elif len(r.shape) == 1:
        return np.concatenate((q[:,0,None]*r[0]-q[:,1:4].dot(r[1:4,None]),
                               q[:,0,None]*r[1:4] + q[:,1:4]*r[0] + np.cross(q[:,1:4], r[1:4], axisa=1)),axis=1)
    elif len(q.shape) == 1:
        return np.concatenate((q[0]*r[:,0,None]-r[:,1:4].dot(q[1:4,None]),
                               q[0]*r[:,1:4] + r[:,0,None]*q[1:4] + np.cross(q[1:4], r[:,1:4], axisb=1)),axis=1)
    else:
        return np.concatenate((np.diag(q[:,0]).dot(r[:,0,None])-np.diagonal(q[:,1:4].dot(r[:,1:
                               np.diag(q[:,0]).dot(r[:,1:4]) + np.diag(r[:,0]).dot(q[:,1:4]) + np.cross(q[:,1:4], r[:,1:4], axisb=1, axisa=1)),axis=1)

def conj_q(q):
    q = array(q, dtype=float)
```

```python
    if len(q.shape) == 1:
        return np.concatenate(([q[0]], -q[1:4]))
    else:
        return np.concatenate((q[:,0,None], -q[:,1:4]),axis=1)

def rot_q(q, v):
    q, v = array(q, dtype=float), array(v, dtype=float)
    if len(q.shape) == 1:
        q /= norm(q)
    else:
        q = np.diag(1.0/norm(q, axis=1)).dot(q)
    if len(v.shape) == 1:
        v = np.concatenate(([0],v))
    else:
        v = np.concatenate((np.zeros((v.shape[0], 1)), v),axis=1)
    v_p = mult_q(q, mult_q(v, conj_q(q)))
    if len(v_p.shape) == 1:
        return v_p[1:4]
    else:
        return v_p[:,1:4]

#Define functions for SLAM operations

def distance_f(y, inliners):
    """distance from line to point function to minimize using least
    squares"""
    theta, rho = y
    return np.abs(np.cos(theta)*inliners.real +
    np.sin(theta)*inliners.imag - rho)

def distance_pair(pair, points):
    """distance from line defined by a pair of points to other
    points"""
    p1, p2 = pair
    return np.abs((p2.real - p1.real)*(p1.imag - points.imag) -
    (p1.real - points.real)*(p2.imag - p1.imag))/np.linalg.norm(p2-p1)

```

```
def EKF(x_hat_minus, Ak, Pk_1, Wk, Qk_1, Hk, Rk, Vk, h_x_hat_minus, zk):
    """Extended Kalman Filter implementation"""
    Pk_minus = Ak.dot(Pk_1).dot(Ak.transpose()) + Wk.dot(Qk_1).dot(Wk.transpose())

    Kk = Pk_minus.dot(Hk.transpose()).dot(np.linalg.inv(Hk.dot(Pk_minus).dot(Hk.transpose()) + Vk.dot(Rk).dot(Vk.transpose()) ))
    x_hat_k = x_hat_minus + Kk.dot(zk - h_x_hat_minus)
    Pk = (np.eye(Pk_minus.shape[0]) - Kk.dot(Hk)).dot(Pk_minus)
    return x_hat_k, Pk

def hi(pk_minus, theta_fi):
    """dhi/dFi"""
    return 1 + (pk_minus.imag*np.cos(theta_fi) - pk_minus.real*np.sin(theta_fi))*1j/2.0

def scan_hi(pk, theta, Fi):
    """Measurement model, remember that theta is real"""
    return Fi.real - (pk.real*np.cos(Fi.imag) + pk.imag*np.sin(Fi.imag) ) + (Fi.imag - theta)*1j

def inv_scan_hi(pk, theta, Fi_hat):
    """Measurement model, remember that theta is real"""
    Th = Fi_hat.imag + theta
    return Fi_hat.real + (pk.real*np.cos(Th) + pk.imag*np.sin(Th) ) + Th*1j

def Ak(theta_k_1, V_flow, dt, N = 0):
    """Ak matrix used for EKF"""
    A = np.eye(4 + N, dtype=complex)
    A[0,1] = np.exp(-theta_k_1*1j)*dt
    A[0,2] = -1j*V_flow*np.exp(-theta_k_1*1j)*dt
    A[2,3] = dt
    return A

```

```
def ransac(points, threshold = 0.15, numSamples = 25):
    """Random Sample Consensus"""
    if points.shape[0] == 0 :
        return None
    if numSamples > points.shape[0]:
        numSamples = points.shape[0]
    choices = np.random.choice(points, (numSamples/2, 2), False)
    best_inliners = np.empty((0,))
    for pair in choices:
        distance = distance_pair(pair, points)
        inliners = points[distance < threshold]
        if inliners.shape[0] > best_inliners.shape[0]:
            best_inliners = inliners.copy()

    #Initialize model with clear inliners
    theta, rho = leastsq(distance_f, [0,1], best_inliners )[0]

    #Re-estimate the model with rescued inliners
    nDistance = np.abs(np.cos(theta)*points.real +
   np.sin(theta)*points.imag - rho)
    theta, rho = leastsq(distance_f, [theta, rho], points[nDistance <
   threshold] )[0]

    return (rho + theta*1j, points[nDistance >= threshold])

def ransac_1p(points, sample, threshold = 0.15, minPnts = 25,
   numSamples = 25):
    """Random Sample Consensus using only one point from the EKF"""
    if points.shape[0] == 0 :
        return None
    if numSamples > points.shape[0]:
        numSamples = points.shape[0]
    choices = np.random.choice(points, (numSamples,), False)
    best_inliners = np.empty((0,))
    for pnt in choices:
        distance = distance_pair([pnt,
   sample.real*np.exp(sample.imag*1j)], points)
```

```
            inliners = points[distance < threshold]
            if inliners.shape[0] > best_inliners.shape[0]:
                best_inliners = inliners.copy()

        if best_inliners.shape[0] < minPnts:
            return None

        #Initialize model with clear inliners
        theta, rho = leastsq(distance_f, [0,1], best_inliners )[0]

        #Re-estimate the model with rescued inliners
        nDistance = np.abs(np.cos(theta)*points.real +
            np.sin(theta)*points.imag - rho)
        theta, rho = leastsq(distance_f, [theta, rho], points[nDistance <
            threshold] )[0]

        return (rho + theta*1j, points[nDistance >= threshold])

#Class used to implement the SLAM algorithm

class slam(object):
    """A class to accomodate the slam algorithm, along with all its
        steps"""

    def __init__(self):
        super(slam, self).__init__()
        #Create and initialize variables
        self.firstScan = True
        self.pose = {'attitude':np.array([1.0,0,0,0]),
            'position':np.zeros((3,))}
        self.mapPnts = []


    def updateStates(self, scanData, odoData, dt = 0.1):
        """Main function to use in order to update state"""
        self.ranges = scanData['ranges']
        goodData = np.logical_not(np.logical_or(np.isnan(self.ranges),
    np.isinf(self.ranges)))
        self.ranges = self.ranges[goodData]
        self.angles = scanData['angles'][goodData]
        self.altitude = odoData['altitude']
        self.velocity = odoData['velocity']
        self.q_imu = odoData['q_imu']
        self.w = odoData['w']
        self.dt = dt

        #Prepare update variables that are going to be used
        self._preUpdate()

        #Check if it's the first scan
        if self.firstScan:

            #Initialize features to track
            self._initFeat()

            #Initialize state vector
            self._initState()
        else:
            #Match features in the expected point
            self._matchFeat()

            #find new features after the match was done
            self._newFeat()

            #Update State
            self._updateState()

            #Add map points to the total point cloud map where they
    should be according to the filters
            self._addPnts2map()

            #Delete repeated features based on distance
            self._delFeatures()
```

```python


    def _preUpdate(self):
        """Prepare update variables that are going to be used"""


        #Put altitude in position
        self.pose['position'][2] = self.altitude

        #Pass scan to a complex representation
        self.complexScan = self.ranges*np.exp(self.angles*1j)

        #Calculate line threshold with 3% of measured distance
        self.threshold = self.ranges*0.03

        #Initialize attitude estimate
        if self.firstScan:
            self.q_hat = self.q_imu

        #Compensate pitch and roll in scan
        self.theta_xy = log_q(self.q_hat)
        self.theta_k = self.theta_xy[2]
        self.theta_xy[2] = 0
        q_xy = exp_q(self.theta_xy)
        self.scanRotated = rot_q(q_xy, np.concatenate((
    [self.complexScan.real],

    [self.complexScan.imag],

    [np.zeros((self.complexScan.shape))]),axis = 0).transpose())

        #Project scan in body frame to ground coordinates
        self.groundScan = self.scanRotated[:,:2]

        #Get velocity
        self.Vflow = self.velocity[0] + self.velocity[0]*1j

        #Get position
        self.complexPos = self.pose['position'][0] +
self.pose['position'][1]*1j

        #Get angular velocity
        self.w_k = self.w[2]


    def _initFeat(self):
        """Initialize features to track"""
        feature = ransac(self.groundScan[:,0] +
self.groundScan[:,1]*1j, self.threshold, 30 )
        self.features = [feature[0]]
        self.featuresPs = [1e-15*(1+1j)]
        for i in range(5):
            feature = ransac(feature[1], np.abs(feature[1])*0.03, 30 )
            if feature[1].shape[0] < 25:
                break
            self.features.append(feature[0])
            self.featuresPs.append(1e-15*(1+1j))
        self.candidateFeat = []

    def _initState(self):
        """Initialize filter matrices"""

        #Intitialize matrices
        self.Qpos = np.diag([1e-8*(1+1j), 0.14603663104771147 +
0.15593997684852676j, 1e-8, 0.0038932125628785971])
        self.Ppos = self.Qpos.copy()
        for P in self.featuresPs:
            self.Ppos = np.diag( np.concatenate((self.Ppos.diagonal(),
[1e-8*(1+1j)])) )
        #Change first scan variable
        self.firstScan = False

        #Create map points
        self.mapPnts = np.empty((0,3))
```

```python

    def _matchFeat(self):
        '''Match features in the expected points'''
        #Pass features to body frame
        fBody = scan_hi(self.complexPos, self.theta_k, np.array(self.features))

        #Search for features in laser scan using 1-point ransac
        self.z_k = []
        points = self.groundScan.copy()
        points = points[:,0] + points[:,1]*1j
        for pnt in fBody:
            feat = ransac_1p(points, pnt )
            if feat:
                self.z_k.append(feat[0])
                points = feat[1]
            else:
                self.z_k.append(None)
        self.remainingPnts = points

    def _newFeat(self):
        '''find new features after the match was done'''
        #Try to match possible new features
        fBody = scan_hi(self.complexPos, self.theta_k, np.array(self.candidateFeat))

        #Search for possible new features in laser scan using 1-point ransac and add them to state variables
        points = self.remainingPnts.copy()
        if points.shape[0] > 25:
            for pnt in fBody:
                feat = ransac_1p(points, pnt, np.abs(points)*0.03 )
                if feat:
                    self.features.append(inv_scan_hi(self.complexPos, self.theta_k, feat[0]))
                    self.z_k.append(feat[0])
                    self.featuresPs.append(1e-8*(1+1j))
                    self.Ppos = np.diag(
np.concatenate((self.Ppos.diagonal(), [1e-8*(1+1j)])) )
                    points = feat[1]
            self.remainingPnts = points

        #Fill new candidate features with remaining points
        if points.shape[0] > 25:
            self.candidateFeat = []
            for i in range(5):
                feature = ransac(points, np.abs(points)*0.03, 30 )
                self.candidateFeat.append(inv_scan_hi(self.complexPos,
self.theta_k,feature[0]) )
                if feature[1].shape[0] < 25:
                    break

    def _updateState(self):
        '''Update State using EKF:
        x_hat_minus, Ak, Pk_1, Wk, Qk_1, Hk, Rk, Vk, h_x_hat_minus,
zk'''
        #Create state vector
        x_hat_minus = np.concatenate(([self.complexPos +
self.Vflow*self.dt,
                                       self.Vflow,
                                       self.theta_k*1j +
self.w_k*1j*self.dt,
                                       self.w_k*1j], self.features))

        A = Ak(self.theta_k, self.Vflow, self.dt, len(self.features))

        #Create Hk
        Hk = np.empty((0,4+len(self.features)))
        h_x_hat_minus = np.empty((0,))
        Zk = []
        numMeas = 0
        for zk, feature, i in zip(self.z_k, self.features,
range(len(self.features))):
            if zk:
                hf = np.zeros((len(self.features),), dtype=complex)
                hf[i] = hi(self.complexPos, feature.imag)
                Hk = np.append(Hk,
    [np.concatenate(([-np.exp(-feature.imag*1j)*0.5, 0, 0.5, 0],
    hf))], axis=0)

                h_x_hat_minus = np.append(h_x_hat_minus,
    scan_hi(self.complexPos, self.theta_k, feature))
                Zk.append(zk)
                numMeas += 1

        Qk = np.diag( np.concatenate((self.Qpos.diagonal(),
    self.features)) )

        Rk = np.eye(numMeas, dtype=complex )*0.03**2

        self.x_hat, self.Ppos = EKF(x_hat_minus, A, self.Ppos,
    np.eye(4+len(self.features), dtype=complex ), Qk, Hk, Rk, np.eye(
    numMeas, dtype=complex ), h_x_hat_minus, Zk )
        self.x_hat[2] = self.x_hat[2].imag*1j
        self.x_hat[3] = self.x_hat[3].imag*1j
        print(self.x_hat)
        print('\n')
        self.pose['position'][0] = self.x_hat[0].real
        self.pose['position'][1] = self.x_hat[0].imag
        self.theta_xy[2] = self.x_hat[2].imag/2
        self.pose['attitude'] = exp_q(self.theta_xy)
        self.q_hat = self.pose['attitude']


    def _addPnts2map(self):
        '''Add scan points to the total point cloud map where they
    should be according to the filters.
        This is just for visualization purposes.'''

        #Compensate rotation in scan with new attitude estimate
```

```
            scanRotated = rot_q(self.q_hat, np.concatenate((
        [self.complexScan.real],

        [self.complexScan.imag],

        [np.zeros((self.complexScan.shape))]),axis = 0).transpose())

            #Pass scan to inertial frame
            scan2Inertial = scanRotated + np.tile(self.pose['position'],
        (scanRotated.shape[0],1))

            self.mapPnts = np.append(self.mapPnts, scan2Inertial, axis=0)

        def _delFeatures(self):
            '''Delete repeated features based on distance.'''
#           tag2del = np.zeros((len(self.features) + 4), dtype=bool)
#           for i in range(len(self.features)):
#               for j in range(i + 1, len(self.features)):
#                   if
        np.abs(self.features[i].real*np.exp(self.features[i]*1j) -
        self.features[j].real*np.exp(self.features[j]*1j)) < 0.15:
#                       tag2del[j+2,0] = True
#                       tag2del[j+2,1] = True

#           tag2del = np.zeros((self.features.shape[0] + 2,2), dtype=bool)
#           for i in range(self.features.shape[0]):
#               for j in range(i + 1, self.features.shape[0]):
#                   if np.linalg.norm(self.features[i] -
        self.features[j]) < 0.2:
#                       tag2del[j+2,0] = True
#                       tag2del[j+2,1] = True
#           self.features = self.features[np.logical_not(tag2del[2:,0])]
#           self.Ppos = self.Ppos[np.logical_not(tag2del.flatten()),:][:,
        np.logical_not(tag2del.flatten())]

if __name__ == '__main__':
    laserData = np.concatenate((np.linspace(-5+0j, -5+5j,
170),np.linspace(-5+5j, 5+5j, 172),np.linspace(5+5j, 5+0j, 170)))
    scanData = {'ranges': np.abs(laserData), 'angles':
np.angle(laserData)}
    odoData = {'altitude': 0.5, 'velocity': [0,0],
'q_imu':np.array([1.0,0,0,0]), 'w':np.array([0.0,0,0]), 'dt':0.1}
    t = slam()
    t.updateStates(scanData, odoData)
    t.updateStates(scanData, odoData)
    t.updateStates(scanData, odoData)
```

Bibliography

Francesco Nex and Fabio Remondino. Uav for 3d mapping applications: a review. *Applied Geomatics*, 6(1):1–15, 2014.

U.S. Air Force. U.S. Air Force air force photos. URL `http://www.af.mil/News/Photos.aspx?igphoto=2000546848`.

Adafruit Industries. 3dr iris - autonomous multicopter -. URL `https://www.flickr.com/photos/adafruit/10578616775/`.

Stefan Winkvist. *Low computational SLAM for an autonomous indoor aerial inspection vehicle.* PhD thesis, University of Warwick, 2013.

Kimon P Valavanis. *Advances in unmanned aerial vehicles: state of the art and the road to autonomy*, volume 33. Springer Science & Business Media, 2008.

Alexander Lanzon, Alessandro Freddi, and Sauro Longhi. Flight control of a quadrotor vehicle subsequent to a rotor failure. *Journal of Guidance, Control, and Dynamics*, 37(2):580–591, 2014.

M Anwar Ma'sum, Grafika Jati, M Kholid Arrofi, Adi Wibowo, Petrus Mursanto, and Wisnu Jatmiko. Autonomous quadcopter swarm robots for object localization and tracking. In *Micro-NanoMechatronics and Human Science (MHS), 2013 International Symposium on*, pages 1–6. IEEE, 2013.

Min-Fan Ricky Lee, Fu Hsin Steven Chiu, and Chen Zhuo. 6 dof manipulator design for maneuvering autonomous aerial mobile robot. In *System Integration (SII), 2013 IEEE/SICE International Symposium on*, pages 173–178. IEEE, 2013.

Mitch Bryson, Alistair Reid, Calvin Hung, Fabio Tozeto Ramos, and Salah Sukkarieh. Cost-effective mapping using unmanned aerial vehicles in ecology monitoring applications. In *Experimental Robotics*, pages 509–523. Springer, 2014.

Robert Oliver. *Robot Localization using Unconventional Sensors.* PhD thesis, 2015.

Suraj Bajracharya. Breezyslam: A simple, efficient, cross-platform python package for simultaneous localization and mapping (thesis). 2014.

Greg Welch and Gary Bishop. An introduction to the kalman filter. *University of North Carolina: Chapel Hill, North Carolina, US*, 2006.

Jack B Kuipers. *Quaternions and rotation sequences*, volume 66. Princeton university press Princeton, 1999.

Simon L Altmann. Hamilton, rodrigues, and the quaternion scandal. *Mathematics Magazine*, pages 291–308, 1989.

Berthold KP Horn. Closed-form solution of absolute orientation using unit quaternions. *JOSA A*, 4(4):629–642, 1987.

Emil Fresk and George Nikolakopoulos. Full quaternion based attitude control for a quadrotor. In *Control Conference (ECC), 2013 European*, pages 3864–3869. IEEE, 2013.

Carlos Izaguirre-Espinosa. Position–yaw tracking of quadrotors. *Journal of Dynamic Systems, Measurement, and Control*, 137:061011–1, 2015.

Xiangke Wang and Changbin Yu. Feedback linearization regulator with coupled attitude and translation dynamics based on unit dual quaternion. In *Intelligent Control (ISIC), 2010 IEEE International Symposium on*, pages 2380–2384. IEEE, 2010.

Herbert Goldstein. *Classical mechanics*, volume 4. Pearson Education India, 1962.

A Alaimo, V Artale, C Milazzo, A Ricciardello, and L Trefiletti. Mathematical modeling and control of a hexacopter. In *Unmanned Aircraft Systems (ICUAS), 2013 International Conference on*, pages 1043–1050. IEEE, 2013.

Pedro Castillo Garcia, Rogelio Lozano, and Alejandro Enrique Dzul. *Modelling and control of mini-flying machines*. Springer Science & Business Media, 2006.

Katsuhiko Ogata and Yanjuan Yang. Modern control engineering. 1970.

Jean-Jacques E Slotine, Weiping Li, et al. *Applied nonlinear control*, volume 199. Prentice-Hall Englewood Cliffs, NJ, 1991.

Stefan Van Der Walt, S Chris Colbert, and Gael Varoquaux. The numpy array: a structure for efficient numerical computation. *Computing in Science & Engineering*, 13(2):22–30, 2011.

Eric Jones, Travis Oliphant, Pearu Peterson, et al. SciPy: Open source scientific tools for Python, 2001–. URL `http://www.scipy.org/`. [Online; accessed 2015-02-05].

James Goppert et al. Python control systems library, 2014–. URL `http://sourceforge.net/projects/python-control/`. [Online; accessed 2015-02-05].

John D Hunter. Matplotlib: A 2d graphics environment. *Computing in science and engineering*, 9(3):90–95, 2007.

Marco Zuliani. Ransac for dummies. *With examples using the RANSAC toolbox for Matlab and more*, 2009.

Javier Civera, Andrew J Davison, and José María Martínez Montiel. 1-point ransac. In *Structure from Motion using the Extended Kalman Filter*, pages 65–97. Springer, 2012.

Rudolf Emil Kalman et al. Contributions to the theory of optimal control. *Boletin de la Sociedad Matematica Mexicana*, 5(2):102–119, 1960.

Greg Welch and Gary Bishop. An introduction to the kalman filter, 1995.

S. Kohlbrecher, J. Meyer, O. von Stryk, and U. Klingauf. A flexible and scalable slam system with full 3d motion estimation. In *Proc. IEEE International Symposium on Safety, Security and Rescue Robotics (SSRR)*. IEEE, November 2011.

Kerry W Spring. Euler parameters and the use of quaternion algebra in the manipulation of finite rotations: a review. *Mechanism and machine theory*, 21(5):365–373, 1986.

Iris Wieser, Alberto Viseras Ruiz, Martin Frassl, Michael Angermann, Joachim Mueller, and Michael Lichtenstern. Autonomous robotic slam-based indoor navigation for high resolution sampling with complete coverage. In *Position, Location and Navigation Symposium-PLANS 2014, 2014 IEEE/ION*, pages 945–951. IEEE, 2014.

Kai M Wurm, Armin Hornung, Maren Bennewitz, Cyrill Stachniss, and Wolfram Burgard. Octomap: A probabilistic, flexible, and compact 3d map representation for robotic systems. In *Proc. of the ICRA 2010 workshop on best practice in 3D perception and modeling for mobile manipulation*, volume 2, 2010.

Inc. Gumstix. *AeroCore MAV Control Board Technical Specifications*, 2014a. URL `http://gumstix.org/images/pdfs/aerocore_technical_specs.pdf`.

Inc. Gumstix. *DuoVero Zephyr Computer-On-Module Technical Specifications*, 2014b. URL `http://media.gumstix.com/docs/DuoVero_Zephyr.pdf`.