POLITECNICO DI TORINO

**Master's Degree
in Mechatronics Engineering**

Master's Thesis

Docking Interface for Aerial-Ground Robot Collaboration: Design and Implementation of an Autonomous Coupling System

Supervisors
Prof. Alessandro RIZZO
Orlando TOVAR ORDOÑEZ
Edoardo TODDE

Student
Eduardo SCHIAVON DE ANDRADE

Academic Year 2024-2025

I am truly grateful to my family and friends for supporting me throughout my academic journey abroad.

I would also like to thank Prof. Alessandro Rizzo, Edoardo Todde, Orlando Tovar Ordoñez, and Andrea Manassi for mentoring me throughout my thesis project at CIM4.0.

Abstract

With the rapid advancement in technology, robotic systems have become increasingly popular and are used to aid in various complex tasks requiring automation. Although UAV and UGV systems have many individual applications, their cooperation has also proven to be of great value to the industry and the scientific community as a whole.

The purpose of this research is to develop a coupling system between a UAV and a UGV so that the drone can land and recharge on the rover. The coupling system is composed of mechanical and electrical interfaces, which aim to correct the drone's landing errors caused by dynamic and environmental uncertainty and recharge the drone's battery.

Initially, a study was carried out to analyze the state of the art of UGV and UAV collaboration. From this analysis, the best trade-off design was chosen, which consisted of a docking station that corrects the drone's landing errors through gravity and two electrodes placed on the base of the station for the recharging of the drone. Then, CAD prototypes were developed in SolidWorks, which were later imported into Gazebo to simulate the landing in a realistic environment with PX4-Autopilot running on the drone's model. The landing was done autonomously by means of a ROS2 node that uses the OpenCV library and a downward-facing camera to acquire an ArUco marker. An error analysis was then carried out to consider how the design would perform in sub-optimal conditions, such as with higher errors in the sensor measurements and with different sizes of the fiducial marker. Lastly, a final prototype was obtained by 3D printing the design.

Contents

List of Tables 4

List of Figures 5

List of Acronyms 7

1 Introduction 9

1.1 UGV and UAV Collaboration . . . 9
1.2 Collaborative Robots and FIXIT . . . 10
1.3 Thesis Objectives and Structure . . . 11

2 State of the Art Research 13

2.1 Issues with Drone Battery Life . . . 13
2.2 Positional Uncertainty in Drone Landing . . . 14
2.3 Literature Review . . . 14
2.3.1 Automated Battery Replacement Station . . . 15
2.3.2 Wireless Charging Station with Active Compensation . . . 16
2.3.3 Contact Charging Station with Passive Compensation . . . 18
2.3.4 AutoCharge Patent . . . 19
2.3.5 Skycatch Patent . . . 21
2.4 Best Trade-off Design . . . 22

3 Theoretical Concepts and Software 23

3.1 Serial Communication Protocols . . . 23
3.1.1 USART . . . 23
3.1.2 SPI . . . 24
3.1.3 I2C . . . 25
3.2 Software Environment . . . 26
3.2.1 PX4 and QGroundControl (QGC) . . . 26
3.2.2 ROS2 . . . 27
3.2.3 Gazebo . . . 28
3.2.4 Software Integration . . . 30

4 Electronics and Firmware of the FIXIT Robot 32
4.1 FIXIT Hardware Components Overview . . . 32
4.1.1 Battery Connectors . . . 33
4.1.2 Battery Management System (BMS) . . . 34
4.1.3 Fuel Gauge Estimator . . . 35
4.1.4 DC-DC Converters and Drone Connector . . . 36
4.1.5 Ethernet Module . . . 37
4.1.6 Microcontroller (uC) . . . 37
4.2 Arduino Code and Command Line Outputs . . . 39
4.2.1 Includes and Instances . . . 40
4.2.2 Setup . . . 40
4.2.3 Main . . . 40
4.2.4 Command Line Outputs . . . 40

5 Mechanical Design 42
5.1 3D Modelling the Docking Mechanism . . . 42
5.2 Design Changes for 3D Printing . . . 44
5.3 Final Design . . . 47
5.4 3D Printing Analysis . . . 48
5.5 From Solidworks to Gazebo . . . 50

6 Simulation Analysis 51
6.1 Installing the Software . . . 51
6.2 Running the Simulation and Explaining the Code . . . 52
6.2.1 Simulation Launch File . . . 52
6.2.2 Autonomous Landing Code . . . 53
6.3 Building Custom Models in Gazebo . . . 54
6.3.1 Custom Vehicle Definition . . . 54
6.3.2 Custom World Definition . . . 58
6.3.3 Adding the Custom Models to PX4 . . . 58
6.4 Simulating Nonideal Scenarios . . . 59
6.4.1 ArUco Marker Size . . . 59
6.4.2 Modified Landing Code . . . 59
6.4.3 Sensor Noise . . . 60
6.4.4 Wind . . . 66
6.4.5 Distance Sensor for Height Measurements . . . 67

7 Conclusion and Future Work 70
7.1 Conclusion . . . 70
7.2 Future Work . . . 71

Bibliography 75

A Code 75

A.1 UWB Sensor Codes 75
A.1.1 uwb_tag.py 75
A.1.2 uwb_anchor.py 77
A.2 Autonomous Landing 78

List of Tables

5.1 3D Printing Parameter Change. 48
6.1 IMU Noise for Simulation. 62
6.2 Default PX4 EKF2 Parameters Sensors 63
6.3 Modified PX4 EKF2 Parameters Sensors 63
6.4 Analysis of Docking Failure with Sensor Noise 69
6.5 Analysis of Docking Failure with Wind 69

List of Figures

1.1 Perseverance and Ingenuity [26]. 10
1.2 Cobot Examples: [14], [28]. 11
2.1 Battery Charging Station [22]. 16
2.2 Charging Station: a) stepper motor, b) slider, c) binary distance laser sensor, d) ultrasonic sensor, e) aluminum frame, f) PVC panel, g) transmitter coil [13]. 17
2.3 Top View of Landing Platform (left), Isometric View of Landing Platform and Legs (right) [15]. 18
2.4 Maximum Landing Error Table (left) and Depiction of Drone's Legs Misalignment (right) [15]. 19
2.5 AutoCharge station connected to the drone [29]. 20
2.6 Illustration of the Gravity Based Docking of the Drone [30]. 21
2.7 Drone Developed at CIM Using F450 Hawk's Work Frame [12]. . . 22
3.1 USART Frame [24] 24
3.2 Single Master and Slave Configuration in SPI [32]. 25
3.3 I2C communication [33]. 26
3.4 Logos of PX4 [5] and QGC [6]. 26
3.5 ROS2 Logo [7]. 27
3.6 Gazebo Logo [3]. 28
3.7 Diagram of the Software Integration. 30
3.8 Micro-DDS Bridge for ROS2-PX4 Communication [4]. 30
4.1 Top view of the PCB [18]. 32
4.2 Block diagram of the PCB [18]. 33
4.3 Mini-Fit Connector of the Pack's Cells [18]. 34
4.4 Functional Block Diagram of the BQ76930 IC [20]. 35
4.5 Schematic of the LTC2944. 36
4.6 Coulomb Counter Circuit and ACR Register. 36
4.7 Schematic of 24V DC/DC Buck Converter [19]. 37
4.8 SPI Master and Slave Configuration. 38
4.9 Example of Command Line Outputs [18]. 41
5.1 Digital Model of the UAV Frame. 42
5.2 Design Improvement by Adding a Support for the UAV's Feet. . . . 43
5.3 Design Improvement by Material Removal. 43

5.4 Design Change To Avoid Supports. 44
5.5 Excess Material Removal. 45
5.6 Zip Tie Hole and 3D Printing. 45
5.7 Updated Design with Two Zip Tie Holes. 46
5.8 Coupling of Zip Tie Hole Designs. 46
5.9 Drone, Landing Gear, and Docking Cone. 47
5.10 Required Printing Supports for Landing Cone Designs. 49
5.11 Required Printing Supports for Landing Gear Designs. 49
5.12 Example of Pose Correction with Blender. 50
6.1 Simulation of Precision Landing using PX4, Gazebo, ROS2 and QGC. 53
6.2 Drone and Docking Mechanism in Gazebo. 56
6.3 Collision Boxes in Gazebo. 57
6.4 Visualization of the Inertias in Gazebo. 57
6.5 Comparison of Filter Performance in Two Scenarios. 64
6.6 Comparison of Filter Performance in Two Scenarios. 65
6.7 Effect of Wind in Gazebo Simulation. 66
6.8 Graph with ROS2 Nodes and Topics. 68

List of Acronyms

UAV	Unmanned Aerial Vehicle
UGV	Unmanned Ground Vehicle
CAD	Computer-Aided Design
PCB	Printed Circuit Board
WPT	Wireless Power Transfer
USART	Universal Synchronous and Asynchronous Receiver-Transmitter
TX	Transmitter
RX	Receiver
LSB	Least Significant Bit
MSB	Most Significant Bit
SPI	Serial Peripheral Interface
SCLK	Serial Clock
MOSI	Master Output Slave Input
MISO	Master Input Slave Output
SCSn	Serial Chip Select
I2C	Inter-Integrated Circuit
SCL	Serial Clock Line
SDL	Serial Data Line
QGC	QGroundControl
GCS	Ground Control Station
MAVLink	Micro Air Vehicle Link
SITL	Software in the Loop
ROS2	Robot Operating System 2
uORB	Micro Object Request Broker
SDF	Simulation Description Format
XML	Extensible Markup Language

STL	Stereolithography
DDS	Data Distributed Services
BMS	Battery Management System
DC	Direct Current
LiPo	Lithium Polymer
IC	Integrated Circuit
ADC	Analog to Digital Converter
NTC	Negative Temperature Coefficient
SoC	State of Charge
ACR	Accumulated Charge Register
MAC	Media Access Control
uC	Microcontroller
EEPROM	Electrically Erasable Programmable Read-Only Memory
ICSP	In-Circuit Serial Programming
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
FDM	Fused Deposition Modeling
GPS	Global Positioning System
IMU	Inertial Measurement Unit
EKF	Extended Kalman Filter
UWB	Ultra-Wideband

Chapter 1

Introduction

1.1 UGV and UAV Collaboration

UAVs contain high-resolution cameras, which makes them useful for a variety of tasks that require accurate observation. They are also capable of quickly reaching areas that are difficult for humans to access. In particular, UAVs have proven to be very useful in the aftermath of disasters since they can access affected areas and look for survivors using an infrared camera. On the other hand, drones have a limited load-carrying capability since a large load decreases the battery life of the vehicle and affects the field of view of the camera. UGVs can perform a variety of ground tasks, which makes them very useful for agricultural, industrial, and spatial applications. Rovers can carry much heavier loads and can be used to perform complex tasks on the ground, but they are slower and have limited visual capabilities [25]. Since these vehicles each have their own trade-offs, UGV-UAV collaboration is being studied to see how their integration improves the overall system. In this regard, the company Competence Industry Manufacturing 4.0 (CIM4.0) has developed a research project called FIXIT, a collaborative robot made up of a drone and a rover designed to be used within the context of industrial automation.

One of the most notable recent uses of a UGV and UAV collaboration was in NASA's Mars 2020 mission. The mission was carried out by two vehicles, namely the Perseverance rover and the Ingenuity drone. The original goal of the joint mission was to successfully deploy Ingenuity and to test its flight in the rarefied atmosphere of Mars. However, as that was quickly achieved, the goal shifted to test how the UAV could aid the rover's exploration with aerial imaging so that the routes could be better planned by avoiding obstacles and finding potential exploration areas [31].

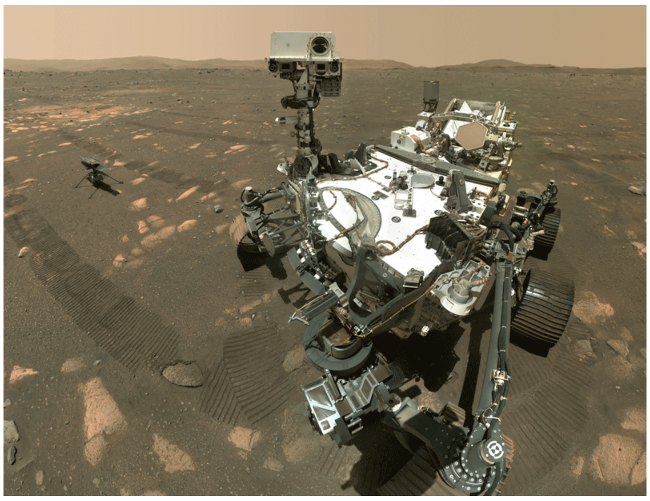
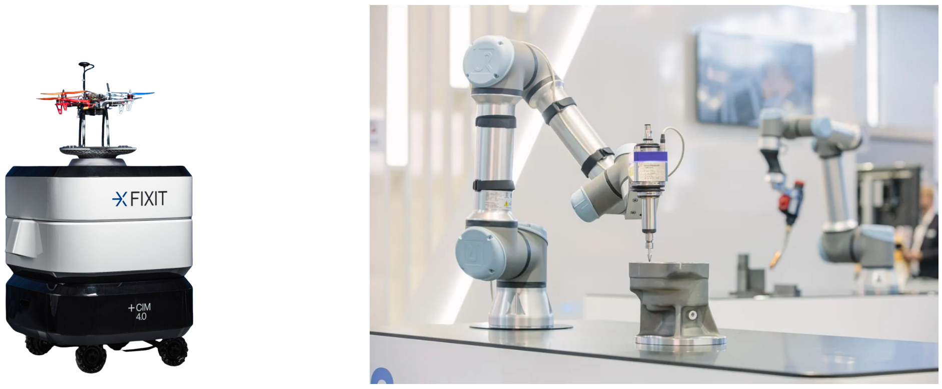


Figure 1.1: Perseverance and Ingenuity [26].

1.2 Collaborative Robots and FIXIT

Industrial robots tend to be isolated from the human workers in the plant since their large size and fast speeds are potentially dangerous for humans [27]. However, a promising trend in industrial automation is to develop robots that can work in unison with workers. These are called Collaborative Robots (Cobots), and their goal is not to replace human activity within the industrial plant but rather to aid the workers in doing their jobs and preventing work-related injuries.

In their earliest stages, cobots were *passive* devices, that is, without any engines so as to increase their operational safety around workers. They would only be employed to help the worker guide an object along a particular path, but without any power of their own, meaning that the operator would still be in control of moving the robot. These types of cobots are now known as Intelligent Assist Devices (IADs), and they paved the way for human-robot collaboration within the industry. Since 2004, however, the focus has shifted to considering *active* cobots such as robotic arms and UGVs.



(a) FIXIT Cobot Developed by CIM4.0. (b) UR5 cobots.

Figure 1.2: Cobot Examples: [14], [28].

FIXIT is a cobot whose goal is to serve as a platform for testing new technologies and expanding knowledge in the industrial automation sector. It consists of a UGV that can guide itself autonomously using sensors and path-planning algorithms to avoid obstacles and a UAV that can cooperate with the UGV [14].

1.3 Thesis Objectives and Structure

This work aims to review the state-of-the-art research on drone-rover collaboration and propose a design that can properly couple a drone to a rover. The best tradeoff solution will be determined from the studied designs, from which we will propose a docking system design. The design will be modeled in SolidWorks and later imported into the Gazebo simulator to study its coupling performance. Finally, non-ideal conditions such as higher sensor error and wind will be considered.

- Chapter 2: covers the state-of-the-art research of electrical and mechanical coupling systems for drone landing. Determines the design that offers the best trade-off among the ones considered in the reviewed literature.
- Chapter 3: explains important theoretical concepts and introduces the software used for running the simulations.
- Chapter 4: introduces and explains the function of the main hardware components present in FIXIT, as well as the Arduino code that runs on the microcontroller of the PCB.
- Chapter 5: describes the design process of the mechanical interface of the chosen model.

- Chapter 6: tests the drone model by importing its CADs into Gazebo and performing different simulations.
- Chapter 7: concludes the thesis' main topic and presents possible design improvements.

Chapter 2

State of the Art Research

2.1 Issues with Drone Battery Life

Drones have proven to be incredibly useful for a variety of different applications. This versatility, coupled with the integrated high-resolution camera technology, has popularized the use of drones to the wider public. The biggest problem that the market faces, however, is the poor autonomy that drones have. The power used by the motors to keep the UAV hovering quickly consumes the drone's battery, and most commercially available drones have a battery life of about 20 to 30 minutes, which can be critical for many applications that require continuous operation.

The simplest solution could be to increase the battery size, yet this proves to be a problem since increasing the battery capacity also means increasing its weight, and a heavier drone spends more energy to hover. Another possible solution would be to manually change the drone's discharged battery to a fully charged one. However, this solution requires a human operator, and in some applications, the drone may be out of reach. Thus, lately, the focus has shifted to considering automated solutions either by using actuated systems to replace the discharged battery [22] or by developing an automatic charging station [13].

This master's thesis proposes to design the mechanical coupling of a UAV and a UGV so that the drone can recharge on the rover in order to increase the drone's flight time.

2.2 Positional Uncertainty in Drone Landing

A popular autonomous landing algorithm for UAVs relies on visual recognition of targets, known as fiducial markers. The visual recognition technique uses software, such as OpenCV, to acquire and manipulate the images obtained by the camera. This technique makes the drone land by first recognizing the target and then estimating the target's pose in 3D based on the 2D image obtained by the onboard camera. The pose is estimated by the internal functions present in the OpenCV library, and once obtained, the UAV can proceed with its landing.

The accuracy of pose estimation depends on factors such as visibility and environmental conditions, sensor measurements, and complex ground effect dynamics. These errors accumulate, producing a landing error in the drone's final position. Since the drone must be recharged after landing, the uncertainty caused by the landing errors must be compensated for because all charging methods require close proximity to the drone.

2.3 Literature Review

Several research papers were analyzed, and most proposed solutions relied on either automating the battery replacement or developing a charging station where the drone could recharge its battery. To determine how to proceed with the design of the coupling system, it was important to understand the system as a whole, from the drone landing to the recharging and then take off.

When it comes to the landing, drones tend to have many inaccuracies, which can be due to the landing code, complex ground effect dynamics, state estimation, sensor errors, and so on. The investigated solutions used either actuated or passive methods to compensate for the landing inaccuracy of the UAV. The actuated methods typically used servomotors that were able to position the charging coil so that it was aligned with the drone [13]. On the other hand, passive methods utilized gravity and contact forces to position the drone [15], [29].

For the recharging, various methods were used, the most common ones being wireless power transfer (WPT) and contact charging. Wireless methods have a considerably lower charging efficiency, which leads to a longer charging time and, therefore, increases the downtime of the drone. Contact charging, conversely, cannot completely remove the downtime of the drone, but it transfers power with a much higher efficiency and, thus, faster charging speeds.

Several commercial solutions have also attempted to tackle this autonomy problem of drones by developing their own automated charging stations. The biggest downsides, however, are the cost and the non-universality of such solutions since they are very expensive and are typically custom-made for specific drone models [1]. A deeper analysis of the studied papers and their proposed solutions is reported successively.

2.3.1 Automated Battery Replacement Station

The solution proposed in [22] uses an automated battery swapping system in order to increase the drone's autonomy while guaranteeing a short downtime. In particular, this solution focuses more on reducing the drone's downtime since it was developed for drone missions with time constraints. Whereas the other innovative approaches of recharging the battery inside the drone are best fit for missions that do not have very stringent time constraints, since the recharging time is on the order of 45 to 60 minutes.

The drone must be placed in the correct position so that the replacement can occur. Therefore, a landing guide was included in the design that provides a landing tolerance of about 15cm, and sliders were placed on the sides so that the UAV can slide into the replacement position. The solution also uses a hot battery swapping feature, which ensures the avionic systems are powered on by an external power supply during the process, allowing the drone to communicate with the ground station during the battery swapping. The total swapping time was designed to be executed within one minute from the drone's landing to takeoff, and the experimental results showed that the process took about 60 seconds. The station holds up to three charged batteries and a depleted one that can be recharged in 45 minutes.

As can be seen from the figure below, an automated battery replacement system has a very complex design, which depends on the well-functioning of many mechanical and electrical components. This means that any source of error in a single component might cause the whole system to malfunction, making it not a robust design. Another downside of this solution is that it is constrained to a specific type of drone since each UAV has its own geometry, and the whole system would have to be redesigned and reprogrammed if it were to be used with a different type of drone. Furthermore, this design was meant to be fixed to the ground, making it nonviable to mount on top of a movable UGV, which is one of the scopes of this thesis.

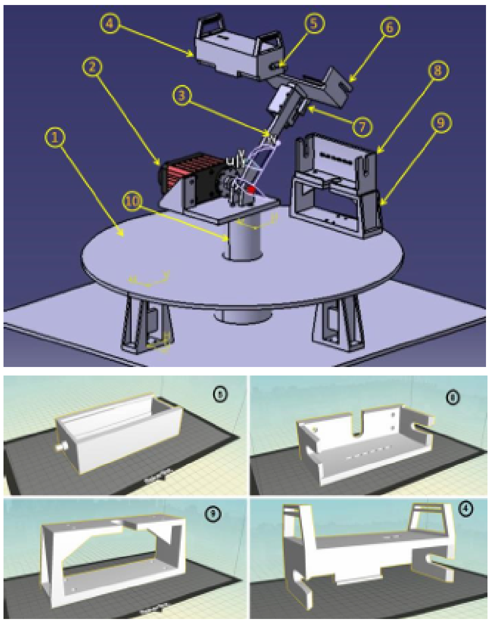


Figure 2.1: Battery Charging Station [22].

2.3.2 Wireless Charging Station with Active Compensation

Wireless Power Transfer (WPT) uses the magnetic coupling between two inductors to transfer power from a primary to a secondary coil. Its working principle is based on Faraday-Lenz's law, which states that a primary coil with a varying magnetic flux can induce a variable current in a secondary coil. Unlike transformers, however, wireless devices do not use a ferromagnetic core to increase the flux flowing through the coils. Instead, the coils transmit this energy wirelessly through the inductive coupling between them. However, this coupling is typically quite weak, producing an inefficient device that can only transfer power over short distances.

Research papers published by MIT [21], [17] showed that using self-resonant coils produced an LC circuit that, when in resonance, was able to transmit power with higher efficiency and over larger distances. This led to the development of LC networks to leverage this property between the inductor coils. Therefore, in most WPT applications, resonant magnetic coupling is used due to its higher efficiency.

The solution proposed in [13] developed an automatic wireless charging station that used sensors to detect the position of the drone, then moved the primary coil to the drone's landing point so that it would recharge the UAV.

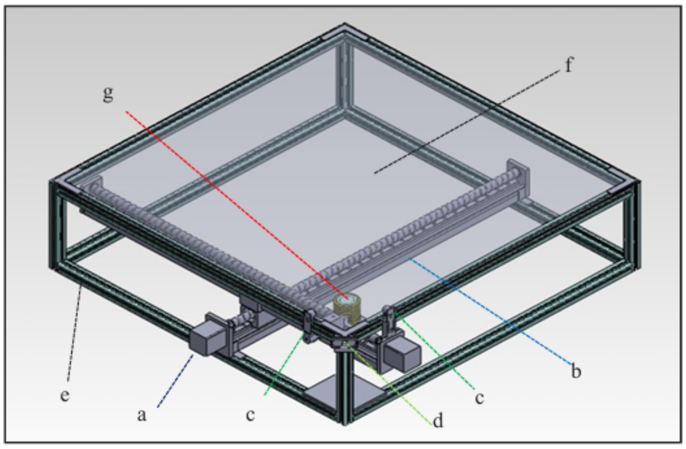


Figure 2.2: Charging Station: a) stepper motor, b) slider, c) binary distance laser sensor, d) ultrasonic sensor, e) aluminum frame, f) PVC panel, g) transmitter coil [13].

The charging station consisted of two stepper motors connected to two linear sliders, a distance laser sensor, an ultrasonic sensor, and the transmitter coil. The working principle is the following: when the drone attempts to land on the station, the ultrasonic sensor is used to determine whether the drone has landed or not. After this initialization stage, the landing position of the drone is determined by binary laser sensors. Once the position of the drone has been determined, the transmitter coil is moved to the position of the UAV by means of the stepper motors, and the charging process may start.

The outcome of the research [13] was that the algorithm used for determining the landing coordinates of the drone was quite accurate, since it determined the position of the drone with an uncertainty of 2mm. The scanning stage, however, took 5 minutes to complete, which corresponds to almost 7% of the total charging time. The charging time was quite long, as it took over an hour for the drone to be charged, and the charging efficiency was quite low because even if the coils were accurately positioned, the efficiency was about 65% that of the wired charging.

The solution proposed in [11] also used wireless charging and an actuated coil positioning device. The alignment algorithm was the main difference with the solution proposed in [13]. In [11], the input impedance between the coils was periodically measured until it reached its predetermined maximum value for a given resonance frequency. The scanning process moved the primary coil based on the measured value of the impedance.

The positive points of wireless charging are its ease of operation and overall safe electronics as it does not leave any exposed wires, making it useful in adverse environmental conditions since the chance of electrode corrosion is much smaller than in contact charging. On the other hand, even in the best-case scenario, wireless systems are inefficient and tend to increase the drone's downtime greatly. The system also depends on the well-functioning of the various sensors and actuators, which increases its complexity.

It was concluded from this research that this type of charging station could be a viable design for this thesis, however, the presence of actuators increases the size of the station making it less portable to be mounted on top of the rover. Additionally, the system depends on the well-functioning of many sensors and electronic systems, which can be additional sources of error.

2.3.3 Contact Charging Station with Passive Compensation

The solution proposed in [15] developed a landing and recharging platform that used a passive method that compensated for the landing inaccuracy of the drone. The UAV uses a vision-based landing algorithm that ensures a landing accuracy of some centimeters. The landing platform is made up of four inverted cones that correct the drone's position by guiding its feet to the center of the cones, which contain slip-ring contacts that recharge the drone's battery.

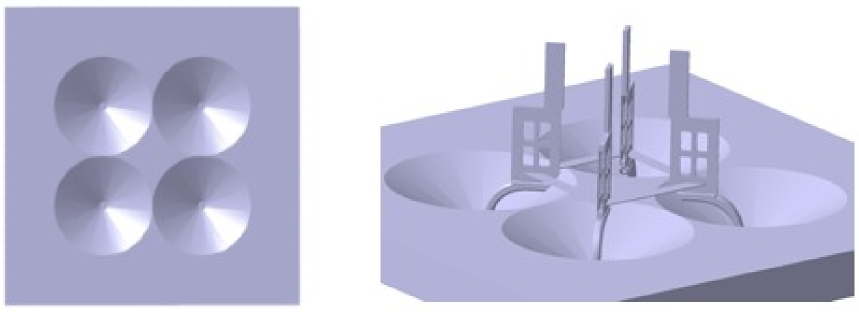


Figure 2.3: Top View of Landing Platform (left), Isometric View of Landing Platform and Legs (right) [15].

The ideal drone position is one where the four legs are all perfectly centered with the center of the cones. However, a generic landing of the drone may include a translational and a rotational component with respect to the ideal position. Therefore, some simulations were performed to estimate the maximum landing error that the platform can correct with a fixed cone radius of 10cm.

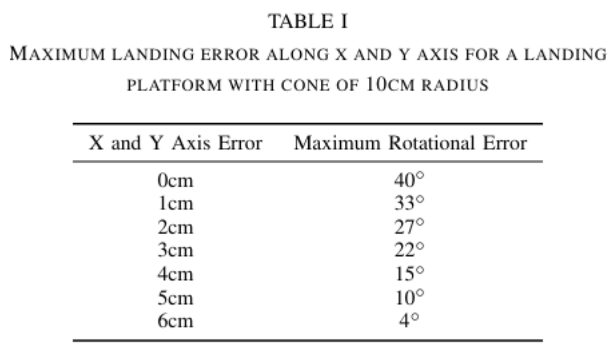


TABLE I

MAXIMUM LANDING ERROR ALONG X AND Y AXIS FOR A LANDING PLATFORM WITH CONE OF 10CM RADIUS

X and Y Axis Error	Maximum Rotational Error
0cm	40°
1cm	33°
2cm	27°
3cm	22°
4cm	15°
5cm	10°
6cm	4°

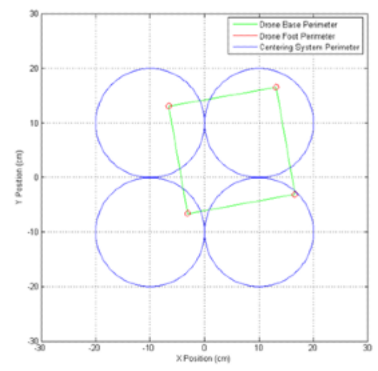


Figure 2.4: Maximum Landing Error Table (left) and Depiction of Drone's Legs Misalignment (right) [15].

As can be seen from the table, the simulations concluded that the maximum landing error the platform is able to compensate for is one of 5cm and 10° of translational and rotational error, respectively. These data were used to design the drone's vision-based landing algorithm. Finally, the research concluded that the drone could land within the maximum error constraints 95% of the time, showing that the overall system offers a reliable solution to the landing inaccuracy problem of drones.

From these results, it can be concluded that passive compensation offers a simple solution to the landing error problem. Furthermore, it greatly simplifies the design of the charging station while not compromising in compensation effectiveness. It is also not difficult to integrate this design with recharging electrodes and the platform and legs can easily be manufactured with a 3D printer.

2.3.4 AutoCharge Patent

The solution proposed in [29] comments on a series of other solutions proposed by different papers. It then compares them to their own developed solution called AutoCharge. To evaluate the different solutions to increase the battery life of drones, the research came up with four parameters, namely, universality, robustness, portability, and efficiency.

The simplest solution to the short battery life would be to increase the battery capacity of the drone. However, as previously mentioned, it is not a true solution to the problem since battery size does not directly increase the flight time of the drone. At a certain point, a larger battery will demand more power from the motors, and its larger capacity will not have a positive effect on the flight time of the UAV.

Battery replacement is the fastest solution from the point of view of the drone's downtime. In fact, the replacement time can be accomplished in just a few minutes, and the drone can immediately return to its previous task. That being said, this solution is often difficult to implement in an autonomous way since it requires many mechanical components. Analyzing this concept with the aforementioned criteria, it was concluded that battery replacement is not a universal solution because it has to be specifically designed for a drone model. It is also not robust since it cannot always overcome landing errors, and it is not portable since the battery-changing stations are large and heavy.

According to [29], contact charging solutions are the most effective, and wireless solutions, appealing as they may be, are quite inefficient from an electrical point of view, and they cause other problems related to alignment and flight control. Therefore, it was concluded that wireless charging is not very efficient nor robust since it requires a precise alignment between the coils in order to achieve efficiencies that are 25-30% inferior to contact charging [13].

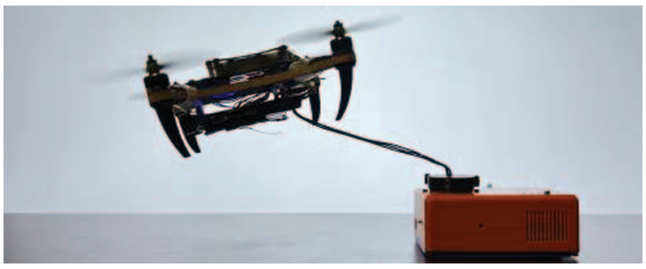


Figure 2.5: AutoCharge station connected to the drone [29].

AutoCharge consists of a small charging station that connects to the drone by means of a tether. At the end of the wire, there is an electromagnet that is able to compensate for drone landing inaccuracies by using magnetic force. Power is fed into the station from the mains, and it is transferred to the drone through an electrode. The magnetic force is only used to attach the electrodes together and is turned off once charging starts.

The research concluded that AutoCharge was able to meet all of the above-mentioned requirements, namely:

- It is Universal since it can be used with any drone
- It is Robust since it compensates for the landing errors
- It is Portable due to its compact size
- It is Efficient because it uses contact charging.

A solution of this type would be viable for this thesis, in particular because of its portability and efficiency. However, the necessity of plugging the station into the mains to recharge the drone restricts its application since, in the case of FIXIT, the station needs to be on top of the rover, which is not always connected to the mains. Moreover, the recharging tether with the magnet would be difficult to employ alongside a visual recognition technique because the camera is typically mounted under the drone, and the presence of the cable could obstruct the camera's field of view.

2.3.5 Skycatch Patent

The patent [30] designed a landing platform along with landing gear capable of compensating for the inaccuracy of the drone's landing through gravity. Once the drone lands on the platform, due to the complementary conic shapes of the landing gear and platform, the weight of the drone causes it to slide down the cone until it reaches its center. The patent also comprises a recharging solution that uses electrodes positioned at the bases of the drone's landing gear and platform. The figure below shows the docking process of a drone when using the proposed design:

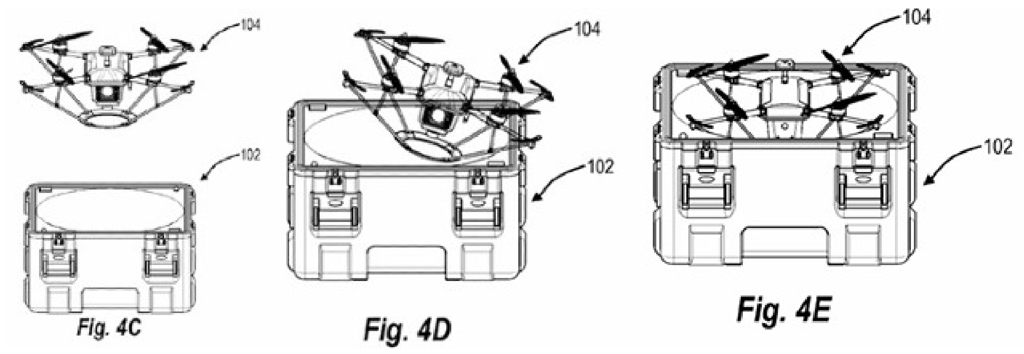


Figure 2.6: Illustration of the Gravity Based Docking of the Drone [30].

It should be noted that a study of the patent was conducted to understand its claims and ensure that we would not infringe on its rights. It was concluded that the original patent was only valid within the territory of the United States and had recently expired, making it possible to use it as inspiration for the design of the recharging station in this Master's Thesis.

2.4 Best Trade-off Design

From the state-of-the-art analysis, we concluded that the best trade-off among the different designs was to develop a charging station with passive compensation and contact charging. As previously mentioned, passive compensation was chosen because it greatly simplifies the mechanical design and offers fewer chances of error due to sensor or actuator malfunction. Contact charging was preferred due to its higher efficiency and simpler implementation.

Regarding the geometry of the station, the adopted design will be similar to [30] and [15]. Our proposed solution will consist of two complementary truncated cone shapes that can be 3D printed in plastic. Initially, the landing gear will be designed to be fixed to the drone's frame. However, the prototype can be further improved to be removable and mounted on various drone models, provided they have the same x-shape.

The drone frame used in the design will be the F450, a commercially available frame that was used in a previous thesis at CIM.

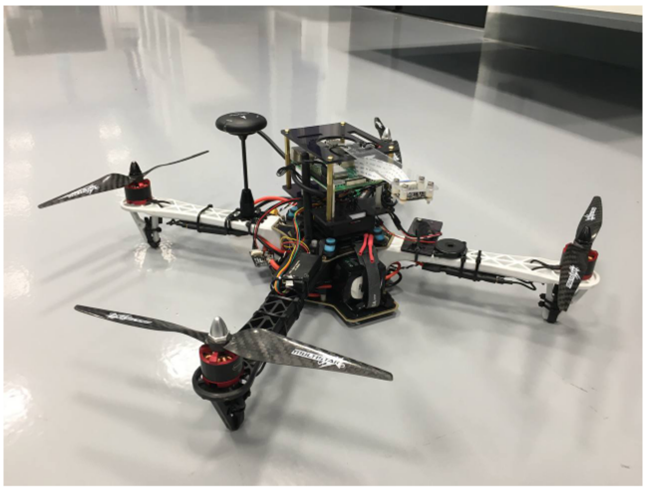


Figure 2.7: Drone Developed at CIM Using F450 Hawk's Work Frame [12].

Chapter 3

Theoretical Concepts and Software

3.1 Serial Communication Protocols

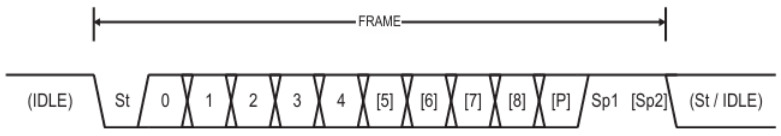
3.1.1 USART

Universal Synchronous and Asynchronous Receiver-Transmitter (USART) is a protocol for sending a continuous stream of data between two types of devices: the transmitter (TX) and the receiver (RX). Two wires are dedicated to data transfer, one for the TX and the other for the RX; thus, USART is full-duplex. This protocol can be configured in both synchronous and asynchronous modes; in synchronous operation, another wire must be used to send the clock signal for the message passing between the TX and RX. Since all data is sent as a continuous stream, the data is codified in frames, which consist of a starting bit (only used for asynchronous operation), data bits, and stop bits. The transmission of data in asynchronous mode works in the following manner:

- When no data is being transferred, the communication line is IDLE, meaning its logic state is set to 1. Electrically, this is configured by means of a pull-up resistor that constantly keeps the line at a high voltage (typically 3.3V or 5V).
- The start bit pulls down the communication line, setting its logical state to 0. Electrically, a logical 0 corresponds to lowering the voltage to GROUND.
- After the starting bit, the data is sent. USART supports a minimum of 5 and a maximum of 9 data bits, which are sent starting from the least significant

bit (LSB) to the most significant bit (MSB). It is also possible to configure a parity bit to increase the success of the data transmission.

- After the data, the stop bit sets the state of the line to a logical 1 by pulling the voltage back up; thus returning to IDLE.



St	Start bit, always low.
(n)	Data bits (0 to 8).
P	Parity bit. Can be odd or even.
Sp	Stop bit, always high.
IDLE	No transfers on the communication line (RxDn or TxDn). An IDLE line must be high.

Figure 3.1: USART Frame [24]

3.1.2 SPI

Serial Peripheral Interface (SPI) is a synchronous protocol that regulates communication between a Master and one or multiple Slaves. SPI uses four wires through which different signals are sent. Since one wire is dedicated to the master output and another to the slave output, SPI offers full-duplex communication:

- SCLK: the wire through which the clock signal is sent.
- MOSI: the wire through which the Master sends data to the Slave.
- MISO: the wire through which the Slave sends data to the Master.
- SCSn: the wire through which the Chip Select is determined. It is required to choose which slave the master will communicate with in a multiple-slave layout.

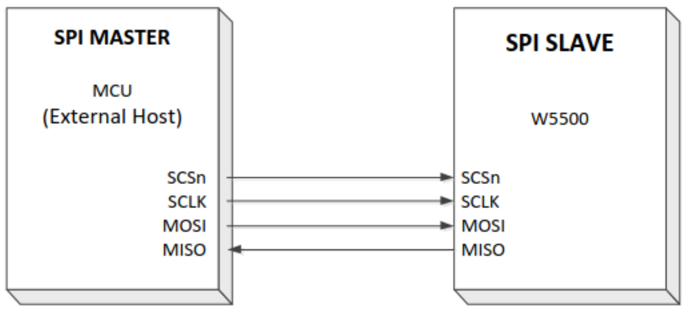


Figure 3.2: Single Master and Slave Configuration in SPI [32].

3.1.3 I2C

The Inter-Integrated Circuit (I2C) is a synchronous protocol that manages serial communication between many devices, called controllers and targets. Unlike USART, however, I2C can also have multiple controllers on the same data bus. This protocol uses two wires: one for transmitting the data, called the serial data line or SDL, and the other for sending the clock signal, called the serial clock line or SCL. Since only one wire is used to transmit data, I2C has a half-duplex transmission. The transmission of data in asynchronous mode works in the following manner:

- Similar to the USART protocol, both lines used in I2C (SDL and SCL) are initially set to TRUE, which is electrically configured by pulling up both lines to a reference voltage.
- For the communication to start, a controller must set the START condition by pulling down the SDL and then the SCL consecutively. This guarantees that only one controller can access the communication channel at a given time.
- Unlike USART, data transmission does not begin immediately after the START condition. Before the data frame is sent, the controller sends an address frame to select the specific target it wants to communicate with. Both frames are made up of 8 main bits, and an acknowledgment bit is used to ensure successful communication. The address frame contains a 7-bit address used to identify the specific target and a final bit that indicates whether the controller will read or write bits to the target.
- After all data frames have been sent, the STOP condition is set by the controller, and it works in a complementary way to the START condition; that is, first the SCL is pulled up and then the SDL.

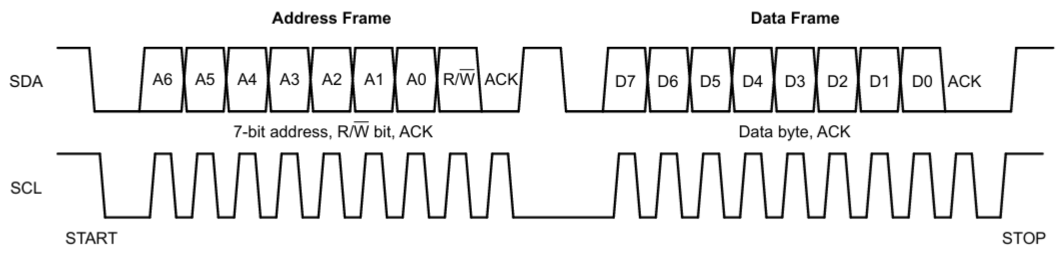


Figure 3.3: I2C communication [33].

3.2 Software Environment

3.2.1 PX4 and QGroundControl (QGC)

(a) PX4-Autopilot.

(b) QGroundControl (QGC).

Figure 3.4: Logos of PX4 [5] and QGC [6].

PX4 is an open-source software that is run on flight controller hardware. It is made up of two main components: the Flight Stack, which is responsible for the control system and the estimation of the variables, and the Middleware, which consists of a communication layer and different sensors. [4]

QGC, on the other hand, is a type of Ground Control Station (GCS) that can send different commands to control the drone, such as takeoff and landing; it can also set a path trajectory for the UAV to follow. The GCS uses the MAVLink Protocol to send its commands to PX4.

A physical drone is controlled by a piece of hardware called a Flight Controller. However, when developing code to control a drone, it is convenient to experiment with it in a realistic simulated environment before running the code on the hardware. This technique is known as Software in the Loop (SITL), and it runs the PX4 firmware on a simulated drone model. Different software can be used for the simulated environment, but this thesis will focus on Gazebo.

3.2.2 ROS2

Figure 3.5: ROS2 Logo [7].

ROS2, which stands for Robot Operating System 2, is an open-source meta-operating system that can be used to create applications for robots. ROS2 is the more updated version of the original ROS1, and it contains some important differences, such as a distributed network architecture. This means that in ROS2, there is no need to establish a master-slave relationship between the different components of the network.

The software works by leveraging the publisher-subscriber mechanism between different components. The main building blocks of ROS2 are the nodes, which are entities that run a specific portion of code that can be used for data processing and robot control, for example. Nodes can interface with each other in three different ways with topics, services, or actions. Topics are mostly used for data that must be sent continuously and monitored periodically, such as sensor data. Services and Actions work based on a request/response communication, and they differ in that actions run for a longer time and can also provide feedback. [23]

- Topic: it is a sort of name tag that identifies the communication between different nodes. Multiple nodes can publish and subscribe to the same topic, and once data is published, all the subscribers receive the transmitted data.
- Service: it is defined by one server and multiple clients. The clients poll the server by sending a request with some data, which is used by the server to perform an operation. After which, the server sends a response to its clients.
- Action: it is characterized by a goal, feedback, and a result. Similarly to services, it interfaces by using clients and a server, however, instead of just receiving one response, the server can also send feedback along the way until the goal is completed.

3.2.3 Gazebo

Figure 3.6: Gazebo Logo [3].

Gazebo[1] is an open-source software that provides a realistic simulated environment in which different types of robots can be operated. Gazebo's internal communication structure is similar to that of ROS2 and is defined in the Gazebo Transport library. The architecture is based on a decentralized network of different nodes that communicate with each other using Gazebo Messages. Although PX4 and Gazebo use different message types, PX4's source code comes with a library called GZBridge, which translates the uORB topics into Gazebo topics and vice versa.

In the context of the SITL feature of PX4, Gazebo is the simulator that contains the drone model along with all of its sensors, and PX4 is responsible for controlling the model. SITL works in the following way: PX4 accesses the sensor data by subscribing to Gazebo sensor topics through the GZBridge library. The acquired data is then processed by the flight controller to obtain the actuator control information. The PX4 node publishes the actuator data, which is then used by Gazebo to move the drone. [4]

```
/world/aruco/model/x500_mono_cam_down_0/link/base_link/sensor
    /air_pressure_sensor/air_pressure
/world/aruco/model/x500_mono_cam_down_0/link/base_link/sensor
    /imu_sensor/imu
/world/aruco/model/x500_mono_cam_down_0/link/base_link/sensor
    /navsat_sensor/navsat
```

[1]In this thesis, Gazebo always refers to the newer version of the software, formerly known as Gazebo Ignition

The code above shows examples of sensor topics to which the Gazebo Simulation publishes its data. PX4 subscribes to these topics to retrieve information about the sensor data. PX4's Flight Stack uses this information to compute the drone's estimated attitude.

```
/model/x500_mono_cam_down_0/command/motor_speed
/model/x500_mono_cam_down_0/servo_0
/model/x500_mono_cam_down_0/servo_1
...
/model/x500_mono_cam_down_0/servo_7
```

These Gazebo topics allow PX4 to publish the actuator control information, such as the motor speed and torque required for the 8 different servo motors of the x500 drone model. Gazebo subscribes to these topics and implements them in the simulation so that the drone may be controlled.

Another powerful feature of Gazebo is the possibility of adding custom models defined in SDF files. These are written in XML format and can be defined for the robotic and environment models in our simulation.[2] From the point of view of the simulation, robot SDF files contain information about the robot's inertial, collision, and visual properties, while world SDF files contain different sensor plugins, and the physical features of the environment such as wind, gravity, magnetic field, atmosphere, etc.

The standard way to define robots in the SDF file is by means of links and joints. Each link contains the inertial properties of the body, its pose relative to the world, its visual features, and its collision box. Joints define a mechanical coupling between a parent and a child link, allowing for relative motion between the two. The file format accepts simple geometries such as a cube, a cylinder, and a sphere to build a robot model. If more complex shapes are needed, however, the visual and collision elements can be imported from a mesh file in STL format.

[2]See [9]

3.2.4 Software Integration

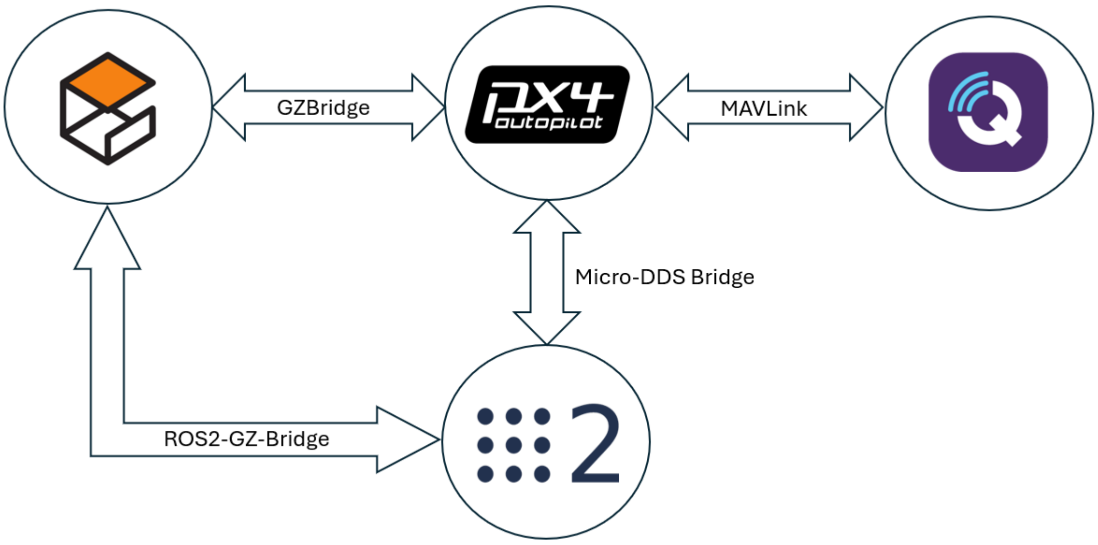


Figure 3.7: Diagram of the Software Integration.

For the purposes of this thesis, all of these various software will be used simultaneously in order to properly control the simulated drone. It must be noted, however, that each software has its own message definitions, and direct communication between two different programs is not guaranteed. Regarding the communication that PX4 has with Gazebo and QGC, PX4's source code already contains the required modules for interfacing with these two software. Despite that, the PX4-ROS2 and Gazebo-ROS2 interfaces cannot communicate without two additional programs, called ROS2-Gazebo Bridge and Micro-DDS.

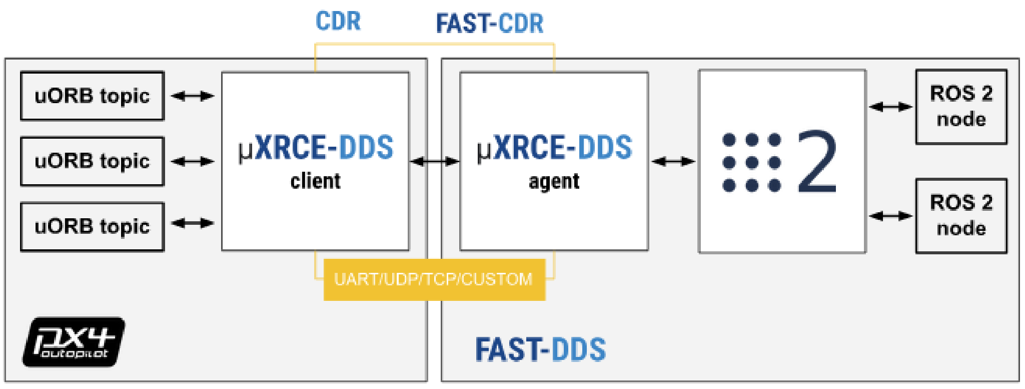


Figure 3.8: Micro-DDS Bridge for ROS2-PX4 Communication [4].

Micro-DDS is the protocol that provides a two-way communication link between PX4 and ROS2, allowing the uORB topics to be accessed by the ROS2 nodes. The Micro-DDS Bridge is automatically managed by software that can be downloaded from the internet. Similarly, the communication link interfacing ROS2 and Gazebo is called ROS2-Gazebo Bridge. It provides a way for ROS2 nodes to access the information in the Gazebo Simulator and vice versa. This two-way communication bridge allows the ROS2 nodes to subscribe to important Gazebo topics, such as topics with sensor data measured during the simulation. [8]

The following two lines of code exemplify how the Gazebo Bridge is used. The syntax indicates that a bidirectional bridge will be created to allow the Gazebo topics */camera* and */camera_info*, which send messages of type *gz.msgs.Image* and *gz.msgs.CameraInfo*, to communicate with the ROS2 topics with message types *sensor_msgs/msg/Image* and *sensor_msgs/msg/CameraInfo*.

```
$ ros2 run ros_gz_bridge parameter_bridge
    /camera@sensor_msgs/msg/Image@gz.msgs.Image
$ ros2 run ros_gz_bridge parameter_bridge
    /camera_info@sensor_msgs/msg/CameraInfo@gz.msgs.CameraInfo
```

Chapter 4

Electronics and Firmware of the FIXIT Robot

4.1 FIXIT Hardware Components Overview

A custom board was previously designed at CIM4.0 to provide all of the required functionalities for the drone and the rover that make up FIXIT. For the scope of this thesis, the most important modules of the board are the battery management system (BMS) and the DC power supply. The BMS performs the proper charging and discharging of the rover's batteries by monitoring cell voltage levels, and it also protects the battery against overvoltage and overcurrent. The power supply module outputs DC power of 5V, 12V, 18V, and 24V by means of several DC-DC regulators.

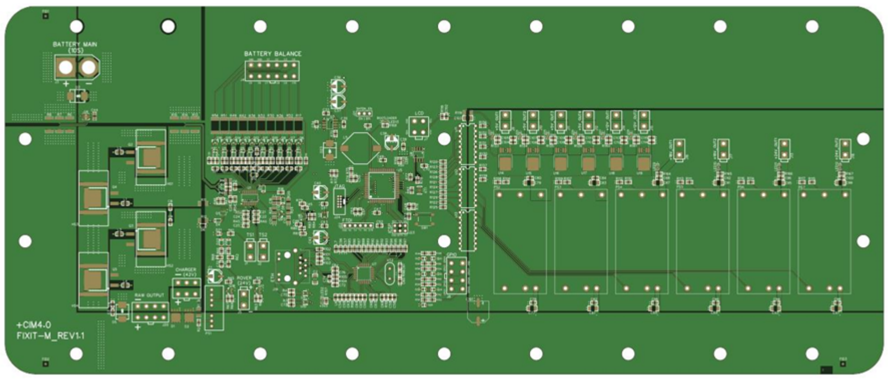


Figure 4.1: Top view of the PCB [18].

In the block diagram below, the grey blocks indicate the different connectors on the board, while the blue blocks indicate the main modules, such as the BMS, the DC-DC regulators, the Ethernet module, and the microcontroller. The Ethernet module allows the user to communicate with the microcontroller, which is responsible for controlling the other modules of the board.

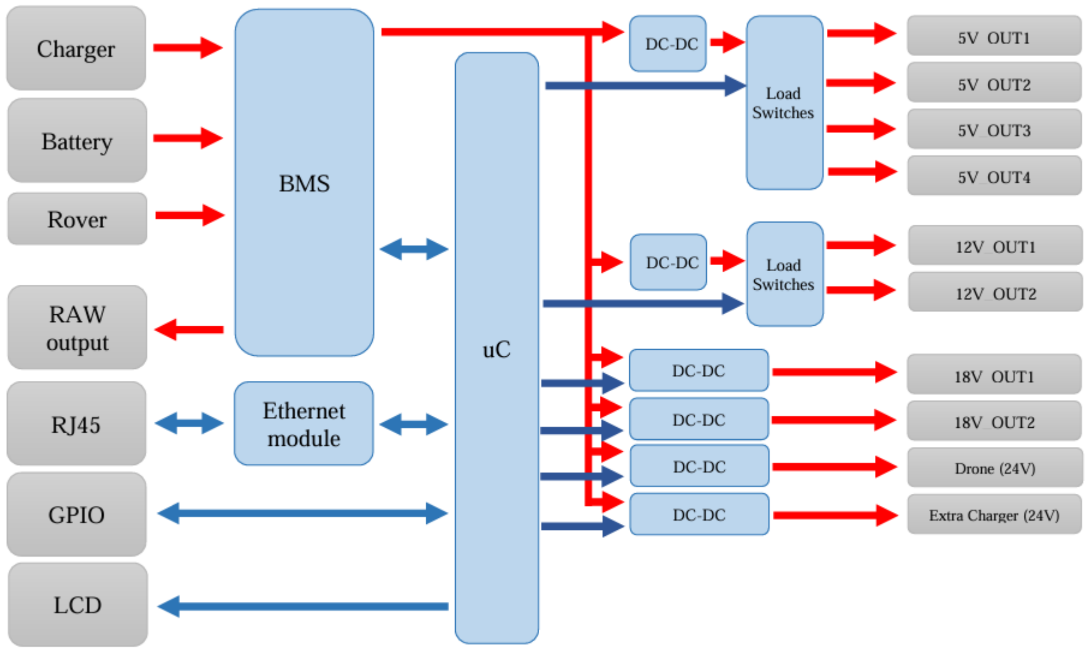


Figure 4.2: Block diagram of the PCB [18].

4.1.1 Battery Connectors

The battery pack used to power the rover is a SAMSUNG 36V 10S2P, comprised of two parallel blocks of 10 cells connected in series. The pack is connected to the board by means of an XT90 connector, which has two pins, one for the positive and the other for the negative poles of the pack. For battery management purposes, each cell is connected to a Mini-Fit connector made up of 12 pins.

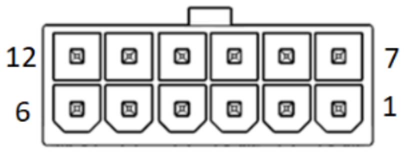
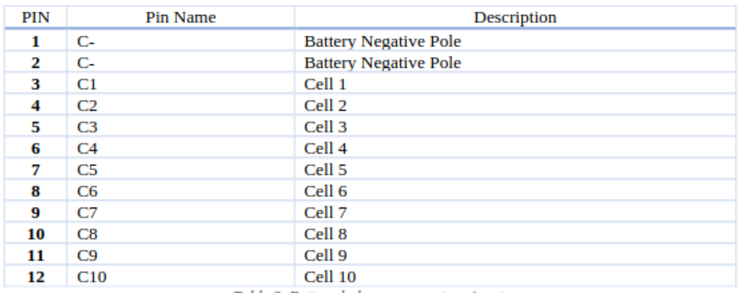


Figure 7. Battery balance connector pinout.



PIN	Pin Name	Description
1	C-	Battery Negative Pole
2	C-	Battery Negative Pole
3	C1	Cell 1
4	C2	Cell 2
5	C3	Cell 3
6	C4	Cell 4
7	C5	Cell 5
8	C6	Cell 6
9	C7	Cell 7
10	C8	Cell 8
11	C9	Cell 9
12	C10	Cell 10

Table 3. Battery balance connector pinout.

Figure 4.3: Mini-Fit Connector of the Pack's Cells [18].

4.1.2 Battery Management System (BMS)

A LiPo battery is typically made up of a series of smaller cells that add up their voltages in order to produce the total output voltage of the battery. In theory, each cell is designed to be equal to the other one in terms of voltage and capacity, but in practice, this cannot be guaranteed since the manufacturing method is not able to account for these exact parameters. Therefore, if we were to charge a battery without a battery management system, it is likely that some of the cells would charge faster than others and would keep increasing their voltage in order to reach the total battery voltage. The end product, in this case, would be a battery with the correct total voltage but unbalanced cell voltages. This is problematic since exposing a battery cell to over-voltage levels might permanently damage the battery.

A battery management system is therefore used for a safe charge/discharge of LiPo batteries since it is able to ensure that all of the cells will charge up to the same maximum voltage without the risk that one of the cells would reach over-voltage levels, and thus decreasing the chance of battery cell damage.

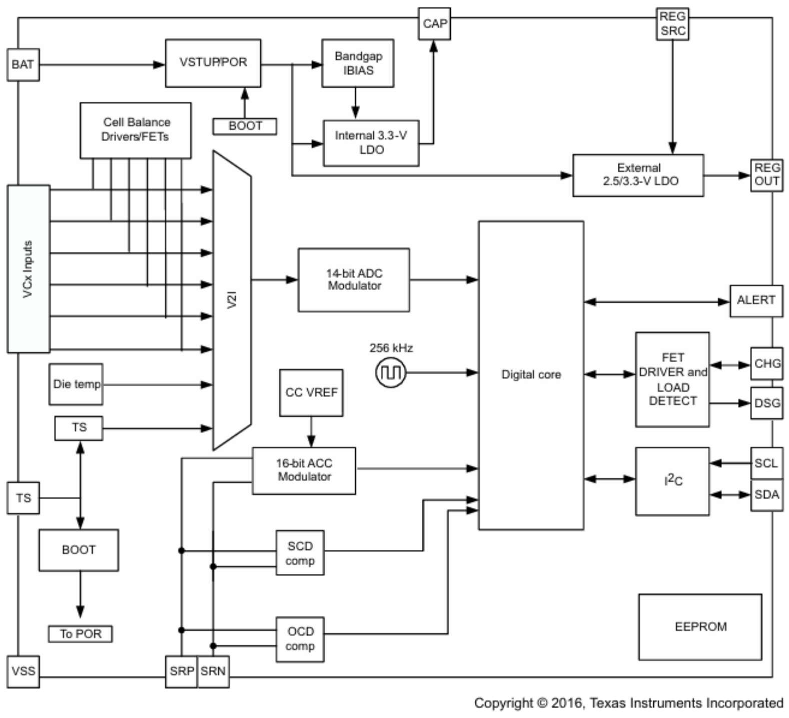


Figure 4.4: Functional Block Diagram of the BQ76930 IC [20].

The BQ76930 chip is responsible for monitoring and balancing the battery cell voltages during charging. In the designed PCB, the BMS makes use of a passive cell balancing system that is controlled by activating a MOSFET, which closes the path for the current to flow through the balancing resistor that dissipates the energy into heat. The cell voltages are read as analog signals and then converted to digital values through a 14-bit ADC. This chip can also monitor the pack's temperature by mounting two NTC thermistors on the appropriate pins. The current flowing through the pack is read by means of a resistor, and its analog value is converted to a digital one by means of a 16-bit integrating ADC. The digital core of the chip processes the acquired data and transmits it to the microcontroller via the I2C protocol through the SCL and SDA pins. [20]

4.1.3 Fuel Gauge Estimator

The LTC2944 chip uses the coulomb counting technique to estimate the battery's state of charge (SoC) and measures the battery's current and voltage. The SoC is the parameter that indicates how charged the battery is, that is, what its current storage capacity is with respect to its full storage capacity, and it is represented as a percentage.

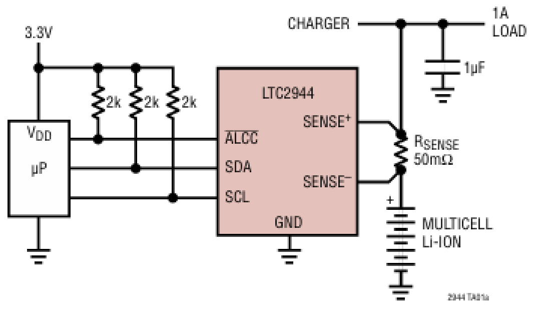


Figure 4.5: Schematic of the LTC2944.

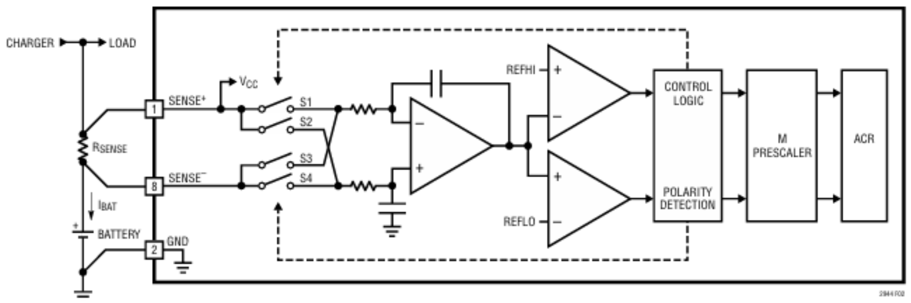


Figure 4.6: Coulomb Counter Circuit and ACR Register.

Fuel Gauge LTC2944 [16].

In the LTC2944 chip, the coulomb counting is performed by an analog circuit (Fig. 4.6) that integrates the current passing through a well-known sense resistor in series with the battery pack. This information is then saved into the accumulated charge register (ACR).

4.1.4 DC-DC Converters and Drone Connector

The board is able to provide a variety of constant DC voltages that can be used to power other systems. For the drone, the 24V/60W output can be used for recharging. The converter used in the PCB is the TEN 60-4815WIN, which is designed to work with an input voltage ranging from 18V to 75V and a constant output of 24V. The CHRG+ pin is either the rectified output of the charger when the rover is being charged or the DC output of the rover's battery. A heat sink (TEN-HS1) can be mounted on top of the converter to help with heat dissipation.

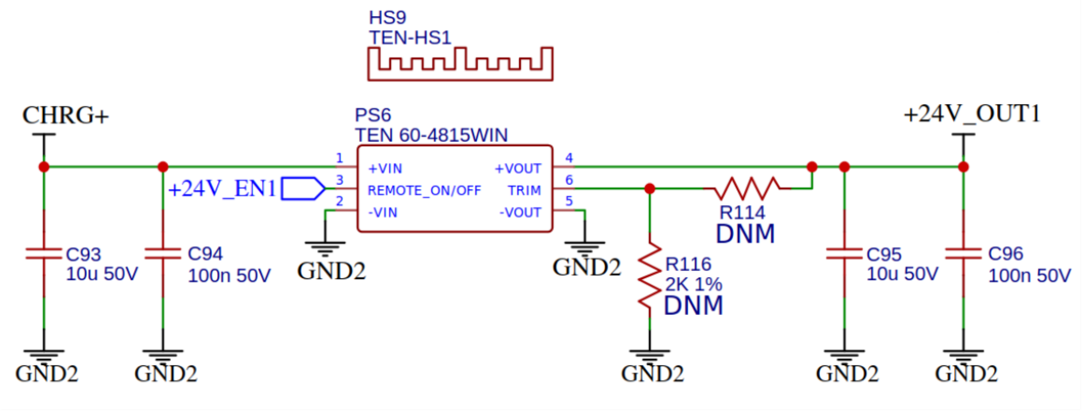


Figure 4.7: Schematic of 24V DC/DC Buck Converter [19].

The board uses Mini-Fit connectors as the output of the DC/DC converters. For recharging purposes, each connector pin of J13 can be wired to the two electrodes located at the base of the docking cone. After landing, these electrodes will come into contact with the drone's electrodes, initiating the recharging process.

4.1.5 Ethernet Module

The Ethernet module is based on three main components, namely, the Ethernet controller (WIZnet W5500), the EEPROM memory (24AA02E48) used to provide the MAC address to the controller, and the RJ45 connector, which links the client to the Wiznet chip. The Ethernet controller has two main interfaces:

- The Media Interface: it is between the RJ45 connector and the W5500 chip, and the connector is electrically decoupled from the Ethernet controller by means of a transformer.
- The SPI Interface: it connects the Host (uC) and the W5500 via the SPI protocol, in which the uC is the SPI Master and the W5500 is the SPI Slave.

4.1.6 Microcontroller (uC)

The "brain" of the board is the Atmel ATmega2560-16AU, which is a high-performance and low-power 8-bit microcontroller. Its main memory components are the flash Memory, the SRAM, and the EEPROM. Both flash and EEPROM memories are non-volatile, meaning that they retain their data even when the uC is powered off. However, their difference lies in the type of data they contain, their storage capacity, and the writing method. SRAM, on the other hand, is a volatile memory that stores the current data being processed by the firmware.

The flash memory can store up to 256KB, and it is dedicated to storing the program that is composed of the bootloader, used to reprogram the uC, and the firmware, which contains the functions developed by the programmer. The flash memory space is divided into two sections, namely, application and boot, which are used for the firmware and the bootloader, respectively [24]. When the board is powered, the bootloader is the first code to run on the uC. It waits for 5 seconds to see if the firmware needs to be updated; if not, it runs the firmware already saved in the flash [18]. The EEPROM memory, on the other hand, can store up to 4KB and saves important data that must be kept even when the board is powered off.

The microcontroller has two main connectors, ICSP and FTDI, through which it can communicate with the computer. The first must be used in order to burn the bootloader onto the board, and the latter is used when loading the firmware. These two connectors communicate with the uC via different protocols: ICSP uses the SPI protocol, and FTDI uses the USART protocol. The ATmega2560 serves as the SPI master for both the W5500 chip and the ICSP controller; therefore, they both share the same MISO, MOSI, and SCK wires. In this specific case, however, each slave has a dedicated CS wire for the uC to manage the communication. The W5500's CS wire (ETH_CS) is connected to the SS pin on the microcontroller, while the CS wire of the ICSP connector is the RESET pin of the microcontroller.

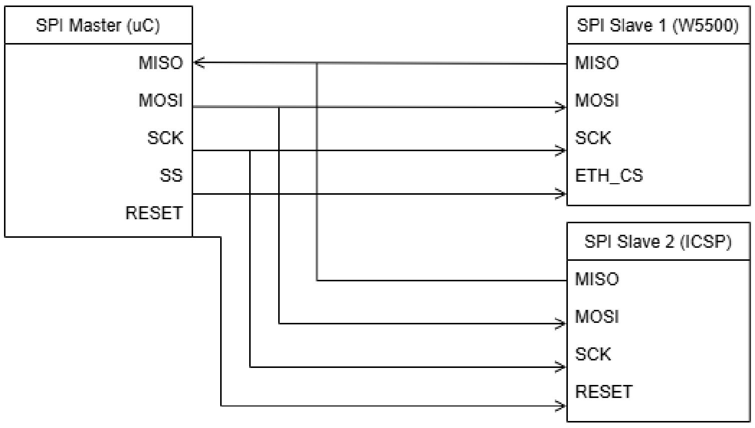


Figure 4.8: SPI Master and Slave Configuration.

The other communication protocol that's used by the uC is I2C. It regulates the data exchange between the microcontroller and the BMS or the Fuel Gauge chips. Although both the fuel gauge and the uC use a 5V voltage level to determine their HIGH and LOW states, the BMS uses a 3.3V voltage level. Thus, an I2C level translator is included in the PCB, and it performs the voltage step-down that's required for the BMS.

4.2 Arduino Code and Command Line Outputs

The main sections of the microcontroller code are reported below:

```
//---- INCLUDES ----//
#include <Wire.h>
#include <EEPROM.h>
#include "rgb_lcd.h"
#include <BQ769x0.h>
#include <LTC2944.h>
#include <Ethernet.h>
#include <SimpleCommandLine.h>
//---- INSTANCES ----//
rgb_lcd           lcd;
SimpleCommandLine CLI;
EthernetClient  TcpClient;
EthernetServer  TcpServer(3001);
LTC2944         GAUGE(GAUGE_RSHUNT_DEFAULT);
BQ769x0         BMS(BMS_PART_NUMBER, BMS_I2C_ADDRESS, BMS_CRC_ENABLED);
//----- SETUP -----//
void setup() {
    Serial.begin(115200);
    InitPins();
    GetHwRevision();
    InitLCD();
    InitEEPROM();
    InitCommandLine();
    InitETH();
    InitBMS();
    GetSerialNumber();
    PrintBoardData();
}
//----- MAIN -----//
void loop() {
    TcpSocket(TcpClient,TcpServer,false);
    TcpCommandPool();
    BmsUpdate();
    GaugeUpdate();
    LcdUpdate();
    BatteryManagement();
    CheckRoverPowerSupply();
    CheckShutdownTimer();
}
```

4.2.1 Includes and Instances

The Arduino code uses libraries written in C++ to access the different data output by the Integrated Circuits of the board. The instances section creates objects of the classes defined in the included header files. For example, the *TC2944.h* file defines the LTC2944 class, from which the instance GAUGE is created in the *FIXIT-M.ino* file. The objects TcpClient/TcpServer(3001) are created from the classes EthernetClient/EthernetServer, defined in the Arduino library *Ethernet.h*. The number 3001 indicates the port used for the TCP communication.

4.2.2 Setup

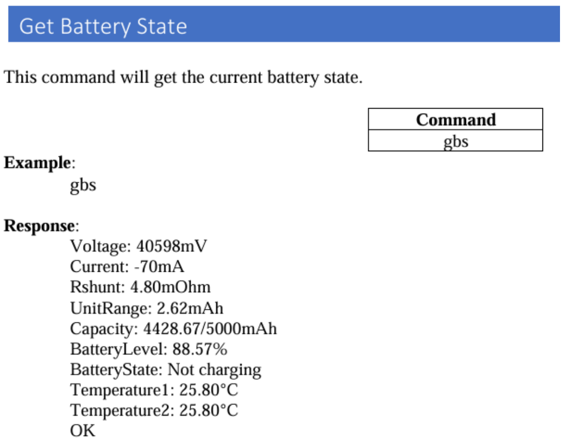
The setup section sets the serial baud rate to 115200 bits per second (bps), which is the data transfer speed to and from the microcontroller when using the USART protocol. This section also initializes the functions required for communication between the rover and the computer, namely, InitETH() and InitEEPROM(). The former initializes the Ethernet device and starts the TCP server, and the latter initializes the board's EEPROM, where the Ethernet module's MAC address is stored. The functions InitBMS() and InitGauge() are used to get the initial information about the battery; they manage the BQ76930 and the LTC2944 chips, respectively. The board also consists of an LCD that displays the cell voltage levels, and therefore, this peripheral is initialized with the function InitLCD().

4.2.3 Main

In the main loop, the TcpSocket() function manages the communication between the client and the server, respectively represented by the computer's command line and the microcontroller. BmsUpdate() updates the BMS information every 50ms, while GaugeUpdate() updates the battery level at a fixed rate of one second set by the GAUGE_UPDATE_INTERVAL constant.

4.2.4 Command Line Outputs

Several commands can be sent from the command line, allowing the user to change specific parameters and informing the user of the state of the battery and its cells. In particular, the useful commands for battery monitoring purposes are *gbs* and *gcs*, whose output is indicated in the figure below. A deeper analysis of the functions contained in the main loop identified that parameters such as battery voltage, current, Rshunt, capacity, and battery level are all obtained from the fuel gauge chip (LTC2944), whereas the battery state and temperature parameters are obtained from the BMS chip (BQ76930). The command *gcs* outputs the information about the voltage of each cell of the rover's battery, which is obtained from the BMS chip.



Get Battery State

This command will get the current battery state.

Command
gbs

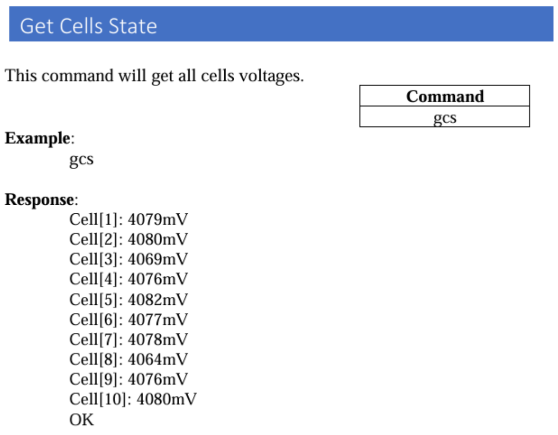
Example:

```
gbs
```

Response:

```
Voltage: 40598mV
Current: -70mA
Rshunt: 4.80mOhm
UnitRange: 2.62mAh
Capacity: 4428.67/5000mAh
BatteryLevel: 88.57%
BatteryState: Not charging
Temperature1: 25.80°C
Temperature2: 25.80°C
OK
```

(a) Output of gbs Command.



Get Cells State

This command will get all cells voltages.

Command
gcs

Example:

```
gcs
```

Response:

```
Cell[1]: 4079mV
Cell[2]: 4080mV
Cell[3]: 4069mV
Cell[4]: 4076mV
Cell[5]: 4082mV
Cell[6]: 4077mV
Cell[7]: 4078mV
Cell[8]: 4064mV
Cell[9]: 4076mV
Cell[10]: 4080mV
OK
```

(b) Output of gcs Command.

Figure 4.9: Example of Command Line Outputs [18].

Chapter 5

Mechanical Design

5.1 3D Modelling the Docking Mechanism

A drone previously built by CIM4.0 on the F450 commercially available frame was used as a model to dimension the docking cone and landing gear. An accurate CAD of the drone was downloaded from the internet [2]:

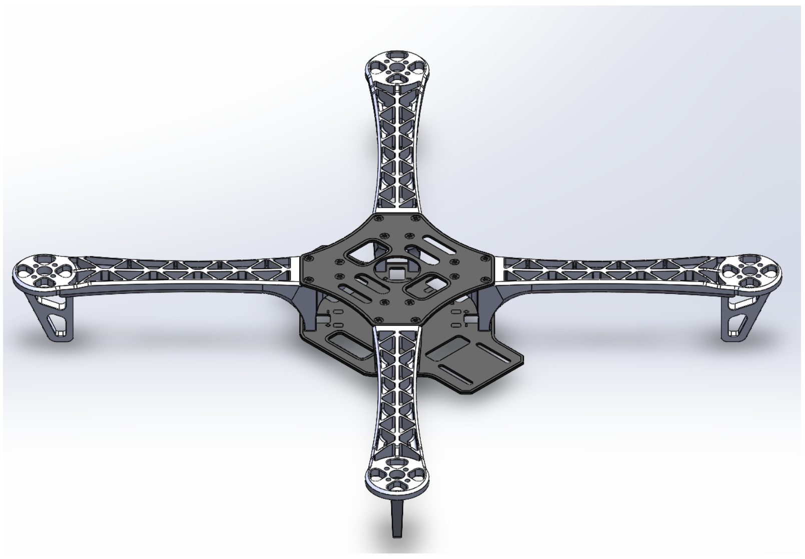


Figure 5.1: Digital Model of the UAV Frame.

The initial design determined the main shape of the landing gear, which consisted of a truncated cone with a base angle of 30°, but it did not include any support for the feet of the drone. The base of the frustum was also too thin, which made it susceptible to high mechanical stresses. The second design, therefore, consisted of a thicker base to ensure better mechanical properties and an opening where the drone's feet could be attached.

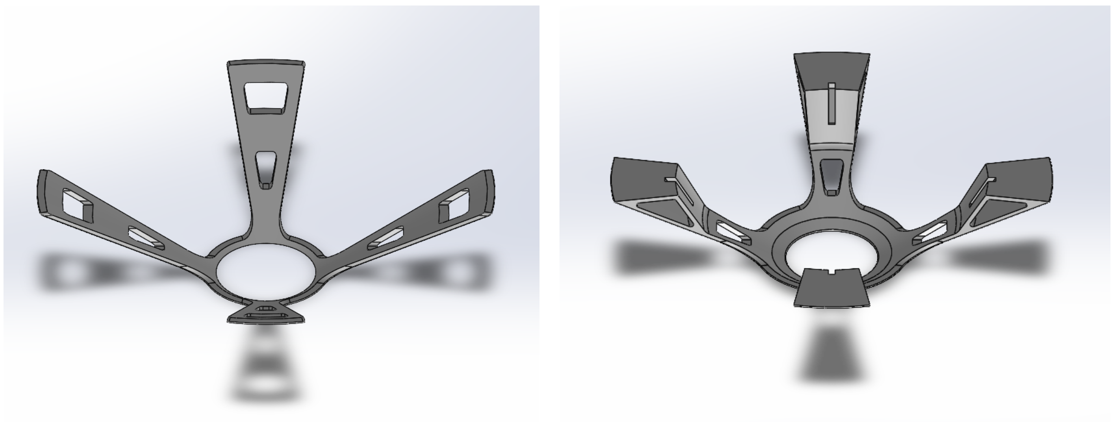

(a) Initial Design. (b) Improved Design.

Figure 5.2: Design Improvement by Adding a Support for the UAV's Feet.

The initial design determined the main shape of the landing cone, which consisted of a truncated cone complementary to the landing gear. The dimensions of the frustum were quite large, and there was material that could be removed without affecting the docking properties. The second design, therefore, consisted of the same shape with added elliptical holes to reduce the amount of material.

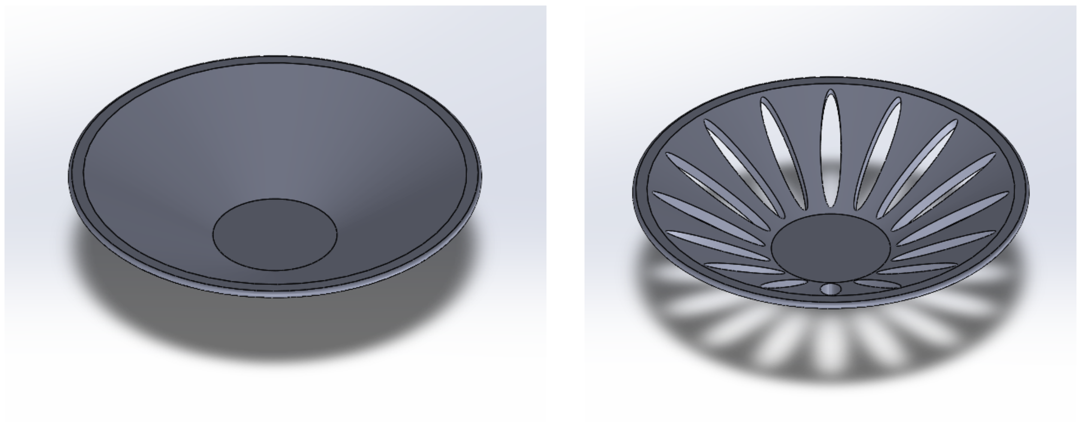

(a) Initial Design. (b) Improved Design.

Figure 5.3: Design Improvement by Material Removal.

5.2 Design Changes for 3D Printing

3D printing an object comes with a series of challenges depending on the object's geometry. In particular, some surfaces cannot be directly printed without adding extra material as support. Such material must be removed later, often producing a rougher surface finish. Another consideration is the amount of material that has to be printed. A fuller material takes longer to print, and it is more expensive. Therefore, two main design changes had to be made:

- Avoid adding unnecessary supports in the printing process.
- Remove material that does not directly interfere with the structure and would prolong the printing time.

Fused Deposition Modeling (FDM) printers deposit plastic filaments layer by layer. Thus, each layer depends on the previous one for support. If a top surface's overhang is larger than 45°, the bottom layer fails to support it, and printing failure will occur if supports are not used.

The initial design of the landing gear had a horizontal surface used to hold the propeller, which would produce an unstable overhang. The following changes were made to avoid adding the printing support:

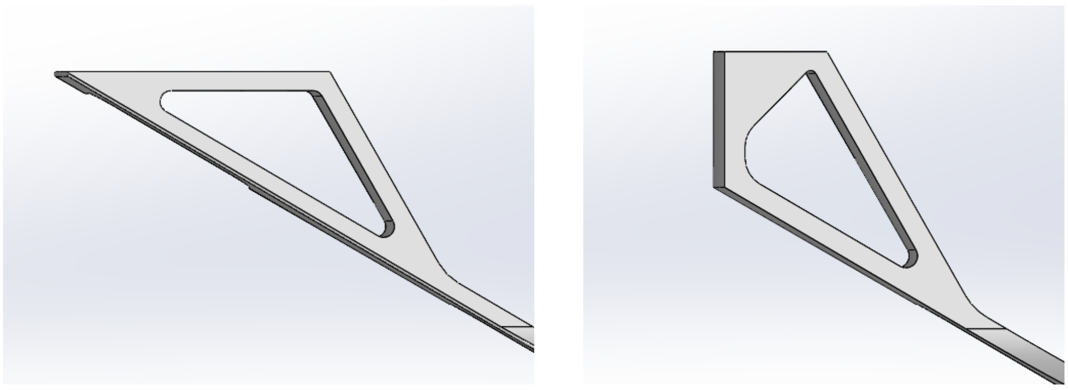
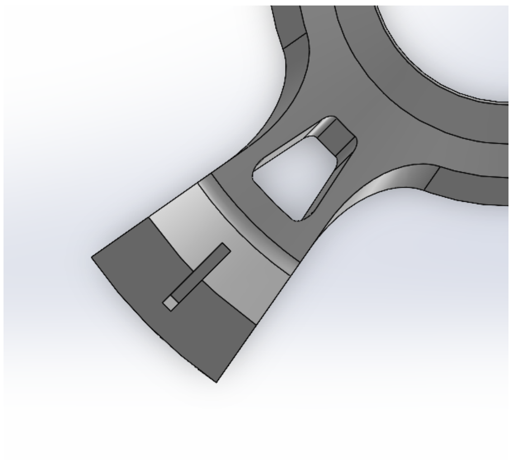

(a) Initial Design. (b) Design after Removal.

Figure 5.4: Design Change To Avoid Supports.

The area in contact with the propeller was larger than required, and it did not interfere with the structure of the design. Therefore, the following changes were made to remove excessive material:


(a) Initial Design.

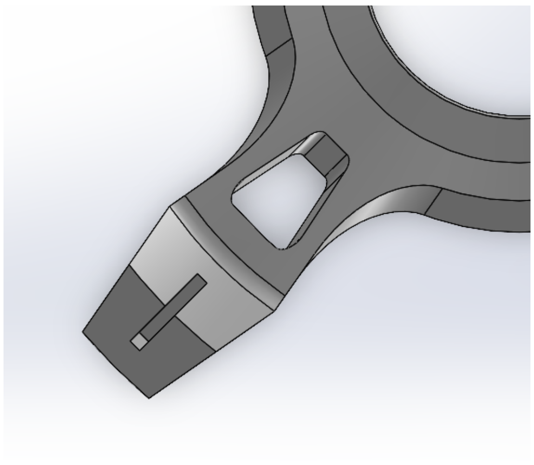
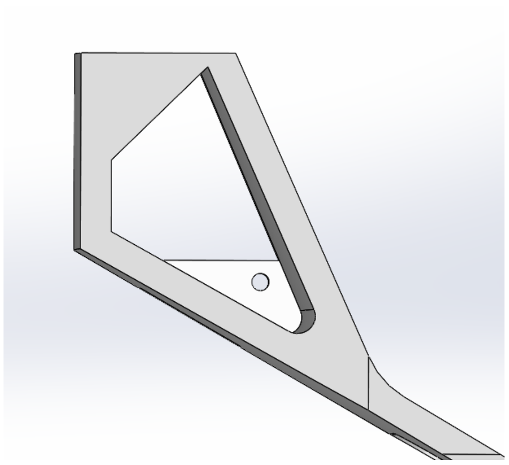

(b) Design after Removal.

Figure 5.5: Excess Material Removal.

A final change was added to fix the drone frame's foot to the landing gear. A small rectangular base was extruded, and a small hole was made to attach a zip tie between the drone's foot and the base. Once these modifications were added to the CADs, a section of the landing gear was printed to test its coupling with the drone's frame. It can be noted that each layer of the outer wall needed to be printed at an angle of 60° with respect to the layer below it, and thus, printing supports had to be used.


(a) Side View of Landing Gear with Zip-Tie Hole.

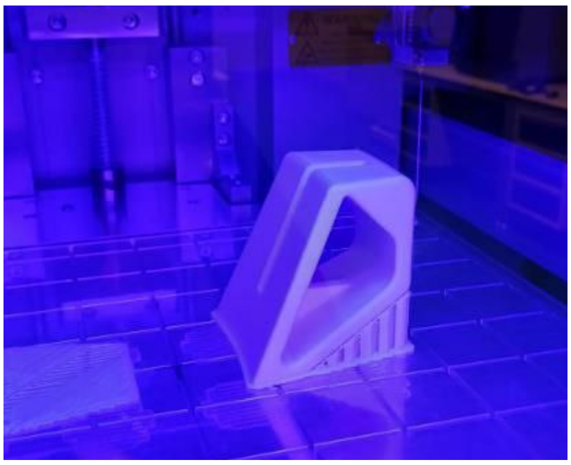
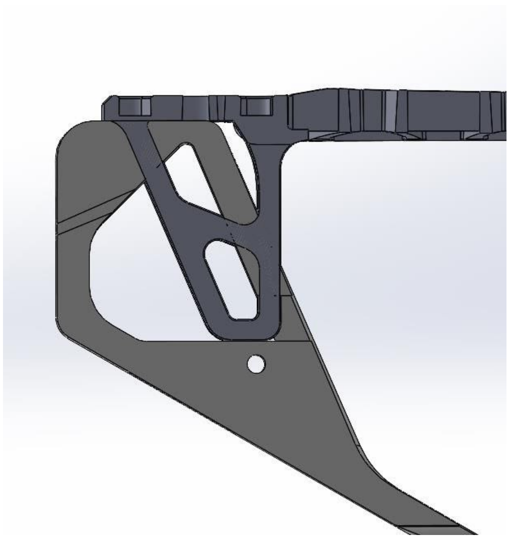

(b) 3D Printing a Section of the Landing Gear.

Figure 5.6: Zip Tie Hole and 3D Printing.

The initial design with one hole for the zip tie proved to be insufficient in tightly coupling the two structures. Therefore, to properly secure the drone's frame, another zip tie hole was added to the exterior wall of the landing gear:


(a) Section View.

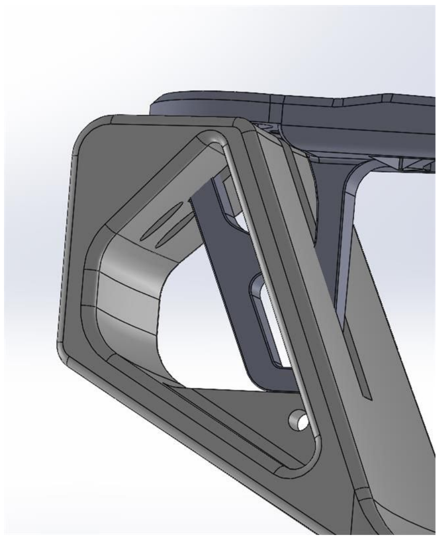
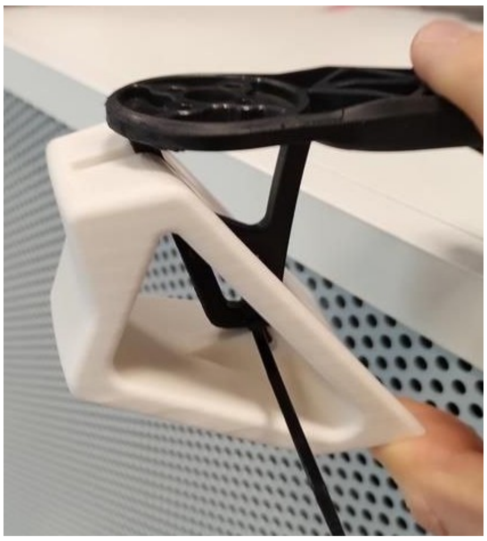

(b) Side View.

Figure 5.7: Updated Design with Two Zip Tie Holes.


(a) Loose Coupling.

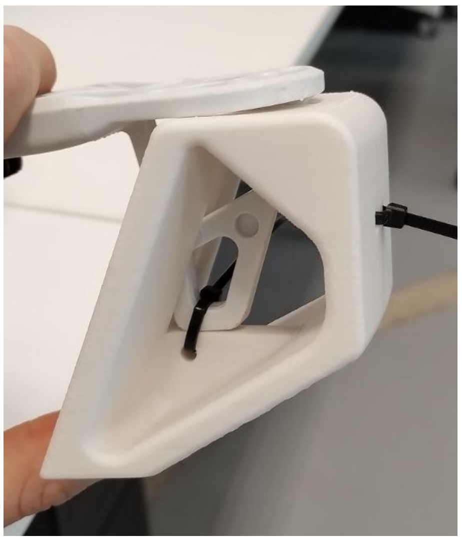
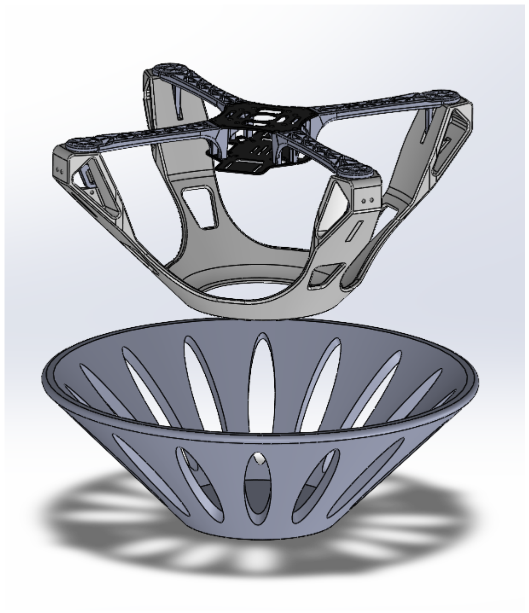

(b) Tighter Coupling.

Figure 5.8: Coupling of Zip Tie Hole Designs.

5.3 Final Design

The angle of the landing gear and cone was changed from 30° to 45°. This choice was made because 30 ° was not large enough to correct the drone's position in the preliminary simulations. With this angle change, some minor changes were also required to adapt the previous design to this new angle, but the overall concept of the design stayed the same. The following figures showcase the final design of the docking mechanism in SolidWorks:


(a) Isometric View.

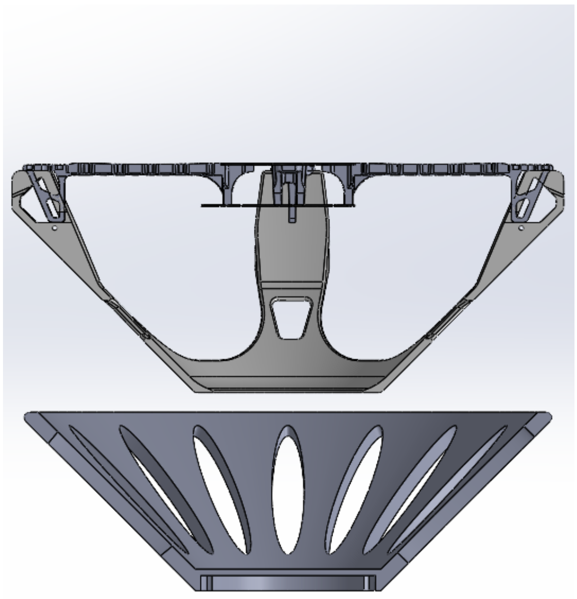

(b) Sectional View.

Figure 5.9: Drone, Landing Gear, and Docking Cone.

By using the *Mass Properties* tool in SolidWorks, it was possible to determine the mass, the moments of inertia, and the center of mass of the objects, three parameters that will be required later for the simulations. The plastic material ABS was chosen as the building material of both components. Moreover, due to the symmetry of the designs, the moments of inertia along the non-principal axes were considered approximately equal to zero.

The mass and moments of inertia of the UAV along its center of mass are:

$$\mathbf{M} = 192.94 \cdot 10^{-3} kg, \mathbf{CM} \approx (0{,}0{,}26.88) mm$$

$$\mathbf{I} \approx \begin{bmatrix} 1869108.65 & 0 & 0 \\ 0 & 1897496.06 & 0 \\ 0 & 0 & 3698030.93 \end{bmatrix} \cdot 10^{-9} \mathrm{kg} \cdot \mathrm{m}^2$$

The mass and moments of inertia of the Landing Gear along its center of mass are:

$$\mathbf{M} = 863.04 \cdot 10^{-3} kg, \mathbf{CM} = (0,0,100.73)mm$$

$$\mathbf{I} \approx \begin{bmatrix} 17924418.92 & 0 & 0 \\ 0 & 17924647.63 & 0 \\ 0 & 0 & 28455943.98 \end{bmatrix} \cdot 10^{-9}\mathrm{kg} \cdot \mathrm{m}^2$$

It can be noted that the largest element of the objects' inertia matrices is Izz, and the other components along the X and Y directions are almost the same. This indicates the object's symmetry about the Z axis.

5.4 3D Printing Analysis

In this section, we will analyze how the design change from 30° to 45° would affect the 3D printing properties. The values of printing time, part volume, and printing support volume reported here were obtained from the 3D printing software predictions.

Part	$V_{part}[cm^3]$	$V_{support}[cm^3]$	$V_{support}/V_{tot}$	Printing Time $[d]$
30° Cone	1495.862	1549.609	51%	3.22
45° Cone	1110.307	43.168	4%	1.44
30° Landing Gear	381.296	334.529	47%	1.15
45° Landing Gear	505.395	170.156	25%	1.42

Table 5.1: 3D Printing Parameter Change.

As can be seen from the table above, a simple design change of the cone angle made a big difference in many printing parameters. In particular:

- The amount of printing supports needed for the cone went from 51% of the cone's printing volume to 4% (93% decrease). Whereas for the landing gear, the amount of printing supports went from 47% to 25% (46% decrease) of the part's printing volume.
- The printing time of the cone was cut by more than 50% while that of the landing gear increased by 24%. However, the total printing time considering both parts of each design decreased by 35%.
- The printing costs were reduced by 39% and the total printing volume of each design decreased by 51%.

As previously mentioned, FDM printers must add supports if the overhang between two layers is larger than 45°. Therefore, we are able to conclude that the final design of the docking mechanism greatly reduced the amount of printing supports required, which also decreased the printing time and costs.

These changes can be visualized in the following pictures, where the parts in green identify the final object and the ones in orange identify the printing supports needed for each design.

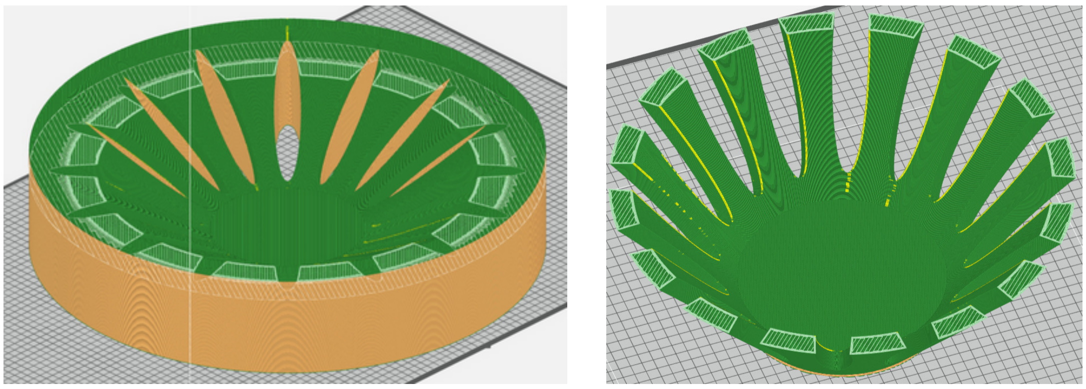

(a) 30° Design. (b) 45° Design.

Figure 5.10: Required Printing Supports for Landing Cone Designs.

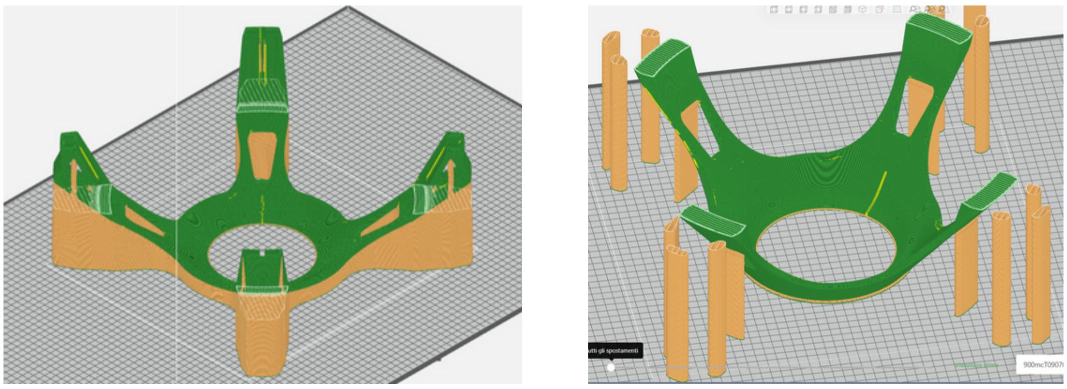

(a) 30° Design. (b) 45° Design.

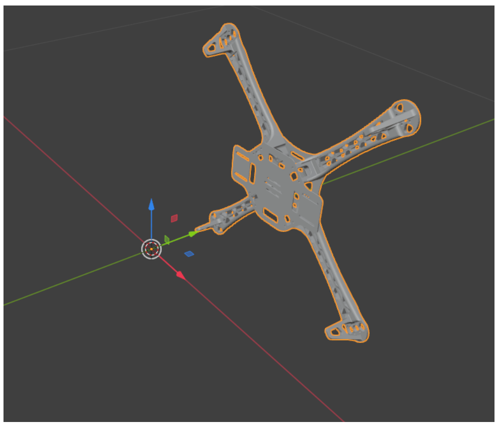
Figure 5.11: Required Printing Supports for Landing Gear Designs.

Another parameter that can be set in the 3D printer software is the amount of material that is added to fill in the object. This reduces the total mass of the designs, since there will be hollow parts inside the final part, which will likely improve the drone's battery life, since it has to carry a lighter weight.

5.5 From Solidworks to Gazebo

The following procedure was done to import the SolidWorks designs into the Gazebo simulation:

- The Solidworks files were saved as STL files.
- The origins of the objects were changed in Blender. This process was not strictly required; however, SolidWorks does not always set the object's origin in the best place for positioning it in the SDF files. Therefore, Blender was used to change the object's origin to a convenient point for assembly.
- The mesh of the object was obtained by exporting the Blender file into an STL format, and it was saved in the UAV model's folder inside the folder: ~/PX4_Autopilot/Tools/simulation/gz/models/fixit_UAV/meshes


(a) Object Imported from SolidWorks.

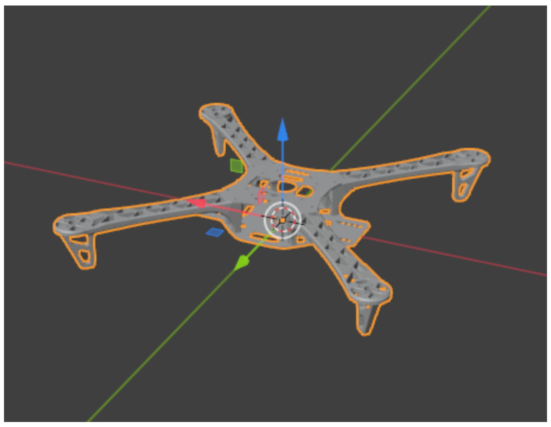

(b) Object with New Origin.

Figure 5.12: Example of Pose Correction with Blender.

Chapter 6

Simulation Analysis

6.1 Installing the Software

All of the software required for the simulation was installed on an Ubuntu 22.04 machine, and the ROS2 distribution used was ROS2 Humble. An open-source code developed by the company ARK Electronics was downloaded from their GitHub repository [10].

1. Installing PX4-Autopilot:

```
$ git clone https://github.com/PX4/PX4-Autopilot.git --recursive
$ bash ./PX4-Autopilot/Tools/setup/ubuntu.sh
$ cd PX4-Autopilot/
$ make px4_sitl
```

2. Installing QGC Daily Build:

```
$ wget
   https://d176tv9ibo4jno.cloudfront.net/builds/master/QGroundControl-x86_64.AppImage
$ chmod +x QGroundControl-x86_64.AppImage
```

3. Installing Micro XRCE-DDS Agent:

```
$ git clone https://github.com/eProsima/Micro-XRCE-DDS-Agent.git
$ cd Micro-XRCE-DDS-Agent
$ mkdir build
$ cd build
$ cmake ..
$ make
$ sudo make install
$ sudo ldconfig /usr/local/lib/
```

4. Installing ROS2-Gazebo Bridge:

```
$ sudo apt install ros-humble-ros-gzgarden
```

5. Cloning the Open-Source Tracktor Beam Repository from GitHub:

```
$ git clone https://github.com/ARK-Electronics/tracktor-beam.git
$ cd tracktor-beam
$ ./install_opencv.sh
$ git submodule update --init --recursive
$ colcon build
```

6.2 Running the Simulation and Explaining the Code

6.2.1 Simulation Launch File

Since the simulation required executing many different terminals simultaneously, a script file was created to execute all of these terminals with only one command. The file was defined as the following:

```
#!/bin/bash
gnome-terminal --tab -- bash -c
    "MicroXRCEAgent udp4 -p 8888; exec bash";\
gnome-terminal --tab -- bash -c
    "./QGroundControl-x86_64.AppImage; exec bash";\
gnome-terminal --tab -- bash -c
    "cd tracktor-beam/; source install/setup.bash;
     ros2 launch aruco_tracker aruco_tracker.launch.py; exec bash";\
gnome-terminal --tab -- bash -c
    "ros2 run rqt_image_view rqt_image_view; exec bash";\
gnome-terminal --tab -- bash -c
    "cd tracktor-beam/; source install/setup.bash;
     ros2 launch precision_land precision_land.launch.py; exec bash";\
```

The first line sets up the Micro-DDS agent and establishes communication between ROS2 and PX4 through the UDP port 8888. The second line starts QGC, and the third line executes the aruco_tracker launch file, which runs the aruco tracker node and sets up the Gazebo bridge for ROS2 to be able to access the /camera and /camera_info topics. Similarly, the fourth line runs the precision_land node. PX4 and Gazebo were launched separately in another terminal.

The initial simulations (as shown in 6.1) were performed with models that were already available in Gazebo to ensure that the code and all of the software were working properly. Having done that, the subsequent simulations were performed with custom drone and world models that will be defined in the following sections.

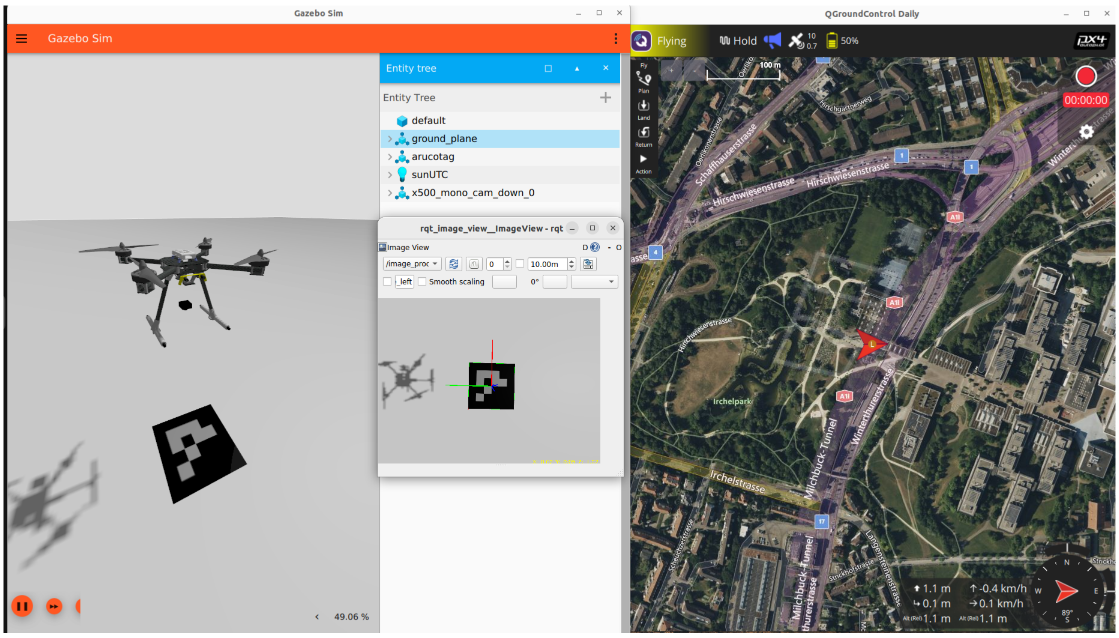


Figure 6.1: Simulation of Precision Landing using PX4, Gazebo, ROS2 and QGC.

6.2.2 Autonomous Landing Code

The downloaded code has two ROS2 nodes called *aruco_tracker* and *precision_land*. The former is responsible for processing the images acquired by the drone's camera, while the latter is responsible for implementing the target search and controlling the drone's landing. The working principles of the nodes are reported below:

- The *aruco_tracker* node subscribes to the Gazebo Topics */camera* and */camera_info* and processes each data in the functions *image_callback()* and *camera_info_callback()*. Then, the marker information and pose estimation are published to the ROS2 */image_proc* and */target_pose* topics.

- The *precision_land* node subscribes to */target_pose* topic that is published by the aruco node. It uses the data received from said topic to compute the target's pose in the world frame, which is then used to change the drone's trajectory so it may land.

6.3 Building Custom Models in Gazebo

6.3.1 Custom Vehicle Definition

In order to have a more accurate simulation of the drone landing with the docking mechanism, a model.sdf file was created based on the vehicle model file of the x500 drone, which comes pre-installed with PX4.

Our drone model is made up of a base link, the landing gear link, four rotor links, and four revolute joints. The base link defines the drone's frame and is connected to the rotor links by the revolute joints. The landing gear link is connected to the drone's frame by a fixed joint since there's no relative motion between the two.

Base Link Definition:

```
<link name="base_link">
  <inertial>
    <mass>0.19294</mass>
    <pose>0 0 0.2688 0 0 0</pose> <!-- COM Position-->
    <inertia>
      <ixx>1869108.65e-09</ixx>
      <iyy>1897496.06</iyy>
      <izz>3698030.93e-09</izz>
    </inertia>
  </inertial>
  <gravity>true</gravity>
  <velocity_decay/>

  <visual name="UAV_visual">
    <pose>0 0 0.110 0 0 0</pose>
    <geometry>
      <mesh>
        <scale>1 1 1</scale>
        <uri>model://fixit_UAV/meshes/UAV.stl</uri>
      </mesh>
    </geometry>
  </visual>
</link>
```

Landing Gear Definition:

```
  <link name="landing_gear">
    <gravity>true</gravity>
    <self_collide>false</self_collide>
    <velocity_decay/>
    <pose>0 0 0 0 0 0</pose>
    <inertial>
        <mass>0.86304</mass>
        <pose>0 0 0.10073 0 0 0</pose> <!-- COM Position-->
        <inertia>
          <ixx>17924418.92e-09</ixx>
          <iyy>17924647.63e-09</iyy>
          <izz>28455943.98e-09</izz>
        </inertia>
    </inertial>
    <visual name="landing_gear_visual">
      <pose>0 0 0 0 0 0</pose>
      <geometry>
        <mesh>
          <scale>1 1 1</scale>
          <uri>model://fixit_UAV/meshes/Landing_Gear.stl</uri>
        </mesh>
      </geometry>
    </visual>

    <collision name="landing_gear_collision">
      <pose>0 0 0 0 0 0</pose>
      <geometry>
        <mesh>
          <scale>1 1 1</scale>
          <uri>model://fixit_UAV/meshes/Landing_Gear_Collision.stl</uri>
        </mesh>
      </geometry>
    </collision>
    </link>
```

Joint Definition:

```
<joint name="landing_gear_joint" type="fixed">
  <parent>base_link</parent>
  <child>landing_gear</child>
</joint>

<joint name="rotor_0_joint" type="revolute">
  <parent>base_link</parent>
  <child>rotor_0</child>
  <axis>
    <xyz>0 0 1</xyz>
    ...
  </axis>
</joint>
```

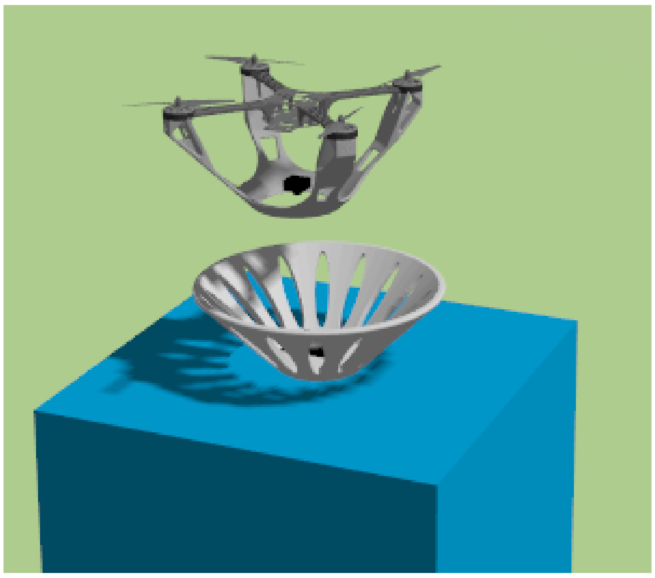
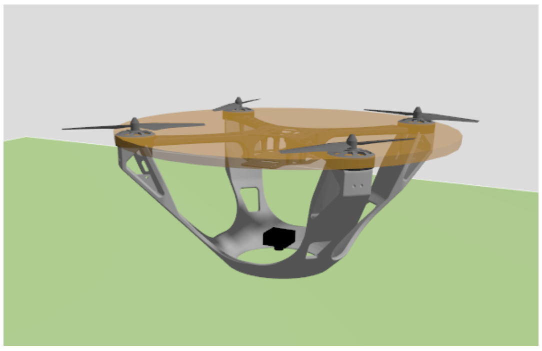


Figure 6.2: Drone and Docking Mechanism in Gazebo.

The drone's visual features and collision boxes were modeled with the meshes of the developed CADs. The inertia values calculated using Solidworks were added to the SDF file and visualized in Gazebo.


(a) Collision Box of the UAV.

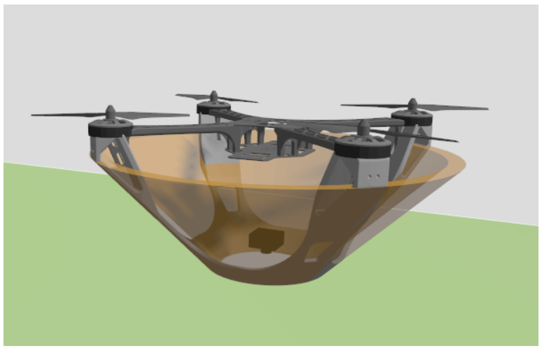
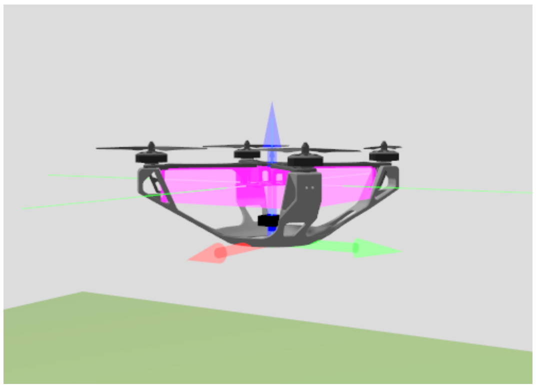

(b) Collision Box of the Landing Gear.

Figure 6.3: Collision Boxes in Gazebo.


(a) UAV's Inertia.

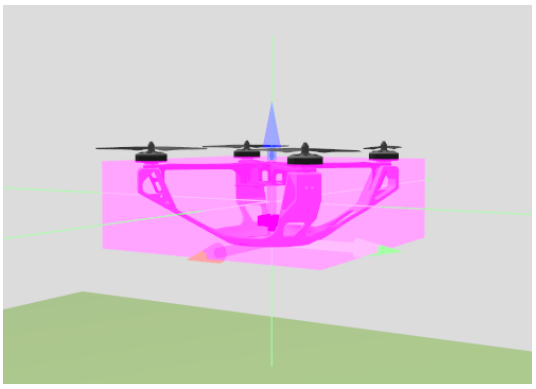

(b) Landing Gear's Inertia.

Figure 6.4: Visualization of the Inertias in Gazebo.

Plugins are also included in the model.sdf file, and they define the rotor models used in the simulation. The plugin *gz-sim-multicopter-motor-model-system* defines the intrinsics of the motors in the model, such as maximum velocity, motor constant, turning direction, etc.

```
<plugin filename="gz-sim-multicopter-motor-model-system"
    name="gz::sim::systems::MulticopterMotorModel">
  <jointName>rotor_0_joint</jointName>
  <linkName>rotor_0</linkName>
  <turningDirection>ccw</turningDirection>
  <commandSubTopic>command/motor_speed</commandSubTopic>
  <maxRotVelocity>1000.0</maxRotVelocity>
  <motorConstant>8.54858e-06</motorConstant>
  <rotorDragCoefficient>8.06428e-05</rotorDragCoefficient>
</plugin>
```

6.3.2 Custom World Definition

The *world.sdf* file is used to define the visual aspects of the simulated environment and also the main parameters behind the simulation, such as the step size, the type of physics engine that is used, and the sensor plugins. In the simulation of this thesis, the world is made up of four main components: the ArUco marker, the UGV, the docking mechanism, and the UAV. The following is an example of the world definition in the SDF file:

```
<world name="fixit">
  <physics type="ode">
    <max_step_size>0.004</max_step_size>
    <real_time_factor>1.0</real_time_factor>
    <real_time_update_rate>250</real_time_update_rate>
  </physics>
  <plugin name="gz::sim::systems::Physics"
     filename="gz-sim-physics-system"/>
  ...
  <plugin name="gz::sim::systems::Imu" filename="gz-sim-imu-system"/>
  <plugin name="gz::sim::systems::AirPressure"
     filename="gz-sim-air-pressure-system"/>
  <plugin name="gz::sim::systems::NavSat"
     filename="gz-sim-navsat-system"/>
  <plugin name="gz::sim::systems::Sensors"
     filename="gz-sim-sensors-system">
  ...
  <include> <uri>model://fixit_aruco</uri> </include>
  <include> <uri>model://fixit_UGV</uri> </include>
  <include> <uri>model://fixit_docking_station</uri> </include>
</world>
```

6.3.3 Adding the Custom Models to PX4

To add our vehicle model, an airframe file must be created containing all of the required parameters of our frame. Then, the model must be added to the CMake file, which contains all of the airframes that can be used in PX4. Each airframe is associated with an ID number, and the ones reserved for custom models are [22000,22999]; thus, the chosen ID for our UAV model was 22000. The main parameters that must be specified in the airframe file are the default simulator, world, and model:

```
PX4_SIMULATOR=${PX4_SIMULATOR:=gz}
PX4_GZ_WORLD=${PX4_GZ_WORLD:=default}
PX4_SIM_MODEL=${PX4_SIM_MODEL:=fixit_UAV}
```

Once the file had been created, it was saved in the folder: ∼/PX4-Autopilot/ROMFS/px4fmu_common/init.d-posix/airframes. The custom world model must be added to the CMake file in the following folder: ∼/PX4-Autopilot/src/modules/simulation/gz_bridge.

A particular command must be used to run the custom files we have built, which specifies the ID of the vehicle that will be launched, the custom world, and an optional third argument that spawns the drone in the desired position:

```
PX4_SYS_AUTOSTART=22000
PX4_GZ_WORLD=fixit
PX4_GZ_MODEL_POSE="0 0 1.041"
./build/px4_sitl_default/bin/px4
```

6.4 Simulating Nonideal Scenarios

6.4.1 ArUco Marker Size

The original aruco_tracker node was designed for an ArUco marker with 50x50 cm^2. In the developed model of the docking mechanism, however, the landing area is constrained by the size of the truncated cone. Therefore, the size of the marker had to be reduced to a 12x12cm^2 square in order to make it fit inside the base. This change in area significantly affects visual recognition since a smaller target is harder to acquire. This generates problems when the drone approaches the target because the landing process will not occur properly if the acquired target is lost. Consequently, a test was carried out in simulation to have a rough estimate of the hovering altitudes that ensure proper landing. The results were that the smaller size of the aruco marker made it only acquirable for altitudes of about 3m or less.

6.4.2 Modified Landing Code

Building on the structure of the code developed by ARK Electronics [10], a new state called *ChangeAltitude* was created to automate the drone's landing. This state changes the drone's altitude if it is higher or lower than a specified threshold value. The *Search* state was modified to switch to the *ChangeAltitude* state once the UAV was close to the base [1]. The ChangeAltitude mode aims to:

- Perform a controlled descent of the drone if it's flying at an altitude higher than 3m with respect to its takeoff. The 3m threshold was determined based

[1]See A.2 for the code

on the experiments mentioned previously.[2]

- Perform a controlled ascent of the drone if it's flying at an altitude lower than 3m with respect to its takeoff. It was decided to have this ascent phase because, in the previous simulations, even if the camera would acquire the target for altitudes lower than 3m, the landing would fail if the drone went directly to its descent phase.

6.4.3 Sensor Noise

The main sensors on the simulated drone are the IMU, the Barometer, and the GPS. The IMU sensor outputs measurements of the drone's acceleration and angular velocities at a high rate of 250 Hz. The Barometer outputs the pressure of the drone during flight at a rate of 50 Hz, and it is used to measure the relative altitude changes of the UAV. The GPS provides the absolute position of the drone at a rate of 30 Hz.

In practice, however, each sensor has noise, which causes the measurements to build up errors in estimating the drone's position. Therefore, the information coming from the sensors is combined in such a way as to reduce the effects that noise has on the control of the drone. This is what's known as sensor fusion, and it is employed in PX4 by means of an Extended Kalman Filter (EKF2), which works in the following way:

- The system is made up of high-rate and low-rate measurements, represented mainly by the IMU and the GPS, respectively.
- The IMU data is fed into a mathematical model containing the drone's dynamic equations. The model then computes a prediction of the system's state, basing itself on the current IMU state.
- Over time, the integration of the IMU measurements causes the predicted state to drift, at which point the observation coming from the low-rate sensors is used to correct the prediction. Then, the predicted state is compared to the observed position measurements taken from the GPS, and the Kalman gain is computed based on the noise of the sensors and on the uncertainty of the predictions.
- Finally, the difference between the prediction and the measurement is corrected by the Kalman gain, producing a corrected state estimate, which is what PX4 uses to control the drone.

[2]See 6.4.1

The sensor plugin definitions in Gazebo allow us to add noise to our simulated sensor values, therefore a Gaussian noise distribution was chosen to simulate the random sensor error. For the IMU, although only the noise in the x-component is shown below, the same values of noise were added to each component. A Gaussian noise was also used for the barometer.

```xml
<sensor name="imu_sensor" type="imu">
  <always_on>1</always_on>
  <update_rate>250</update_rate>
  <imu>
    <angular_velocity> <--! Gyroscope Noise-->
      <x>
        <noise type="gaussian">
          <mean>0</mean>
          <stddev>0.00018665</stddev>
          <dynamic_bias_stddev>3.8785e-05</dynamic_bias_stddev>
          <dynamic_bias_correlation_time>1000</dynamic_bias_correlation_time>
        </noise>
      </x>
    </angular_velocity>
    <linear_acceleration> <--! Accelerometer Noise-->
      <x>
        <noise type="gaussian">
          <mean>0</mean>
          <stddev>0.00186</stddev>
          <dynamic_bias_stddev>0.006</dynamic_bias_stddev>
          <dynamic_bias_correlation_time>300</dynamic_bias_correlation_time>
        </noise>
      </x>
    </linear_acceleration>
  </imu>
</sensor>
<sensor name="air_pressure_sensor" type="air_pressure">
  <always_on>1</always_on>
  <update_rate>50</update_rate>
  <air_pressure>
    <pressure>
      <noise type="gaussian">
        <mean>0</mean>
        <stddev>0.01</stddev>
      </noise>
    </pressure>
  </air_pressure>
</sensor>
```

The default values that were used in the original Gazebo x500 drone model were quite low. Thus, three main scenarios were considered in the simulation: high noise, medium noise, and low noise, which roughly correspond to three price tags of IMU sensors.

Sensor σ	Low Noise	Medium Noise	High Noise
Gyro σ	0.0001 - 0.0005	0.0005 - 0.005	0.005 - 0.02
Acc σ	0.005 - 0.02	0.02 - 0.1	0.1 - 0.5

Table 6.1: IMU Noise for Simulation.

After running the simulations, it was observed that although the IMU noise was increased, it did not significantly affect the simulation's performance. This is because the individual sensor noise and its effects on the estimator algorithm must be considered separately. That is, the Extended Kalman Filter (EKF2) associates its own weight to each of the measured values, which indicates the reliability of that data to the final estimated output. These weights are determined by the filter parameters, which can be monitored in QGC:

- EKF2_ACC_NOISE $\rightarrow$ Corresponds to the predicted standard deviation of the accelerometer.
- EKF2_GYR_NOISE $\rightarrow$ Corresponds to the predicted standard deviation of the gyroscope.
- EKF2_BARO_NOISE $\rightarrow$ Corresponds to the predicted standard deviation of the barometer.
- EKF2_GPS_NOISE $\rightarrow$ Corresponds to the positional error of the GPS.
- EKF2_ACC_B_NOISE $\rightarrow$ Predicted sensor drift associated with the accelerometer.
- EKF2_GYR_B_NOISE $\rightarrow$ Predicted sensor drift associated with the gyroscope.

When compared to the noise values that were being used in the simulations, it can be noted that the EKF2 parameters for these sensors are much higher. This means that the PX4 flight controller expects the IMU and barometer data to be quite noisy and does not trust them much, relying mainly on the GPS data for its pose estimation. This explains why docking was achieved even with higher IMU and Barometer errors.[3]

[3]See 6.4.3

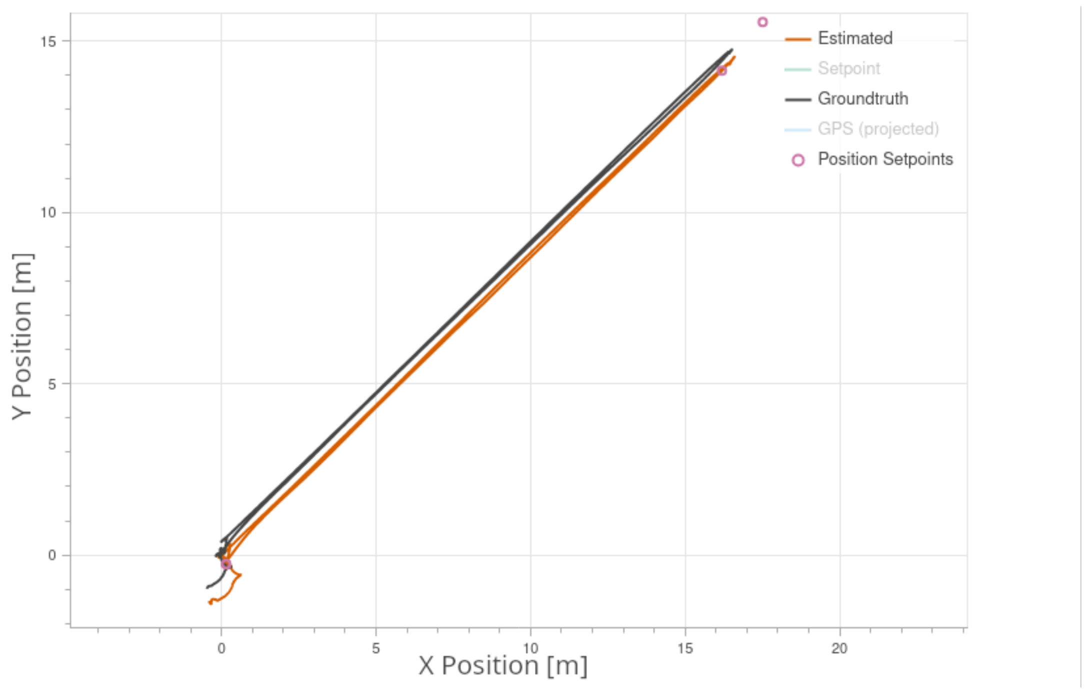
Parameter	Default Value	Units
EKF2_ACC_NOISE	0.35	m/s^2
EKF2_GYR_NOISE	0.015	rad/s
EKF2_GPS_NOISE	0.5	m
EKF2_BARO_NOISE	3.5	m
EKF2_ACC_B_NOISE	0.003	m/s^3
EKF2_GYR_B_NOISE	0.001	rad/s^2

Table 6.2: Default PX4 EKF2 Parameters Sensors

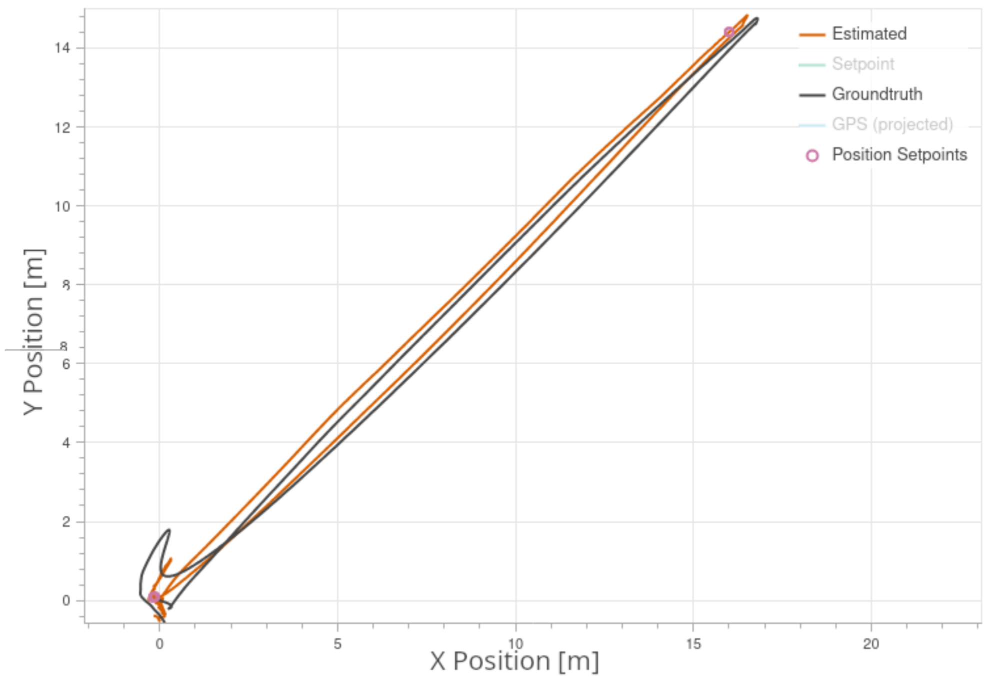
To further understand how the value of the EKF2 parameters affects the drone's control, the parameters of the IMU sensor were changed to values closer to the simulated noise. This caused the flight controller to output improper state estimations since the low value of the parameter put too much trust in the state predictions and not enough on the observer corrections. Thus, when the prediction started to diverge from the observed measurement of the GPS, the filter's corrections did not carry enough weight to adjust the estimated state. Figure 6.5 shows the simulated path of the drone along the XY-plane and the estimated states coming from the EKF2. It can be noted that although the estimates diverged from the groundtruth in the case with modified parameters, the error was not that large, meaning that the drone was still able to follow the position setpoints. The reason why the groundtruth measurements also change with the estimate is that they both depend on each other, since PX4 publishes actuator commands based on the estimates obtained by fusing the Gazebo sensor data. Figure 6.6 shows the altitude estimate and raw GPS and barometer data. In the case with modified filter parameters, the fused altitude estimation greatly diverged from the measured sensor data. After 0:30, the estimator was outputting a state that was above the setpoint, which caused the flight controller to reduce the thrust on the drone, and that caused it to go down to 1.5m. Only after 0:40 did the flight controller react to the altitude difference by increasing the thrust, which caused the drone's altitude to increase after 1:00.

Parameter	Modified Value	Units
EKF2_ACC_NOISE	0.01	m/s^2
EKF2_GYR_NOISE	0.001	rad/s
EKF2_ACC_B_NOISE	0.005	m/s^3
EKF2_GYR_B_NOISE	0.0005	rad/s^3

Table 6.3: Modified PX4 EKF2 Parameters Sensors

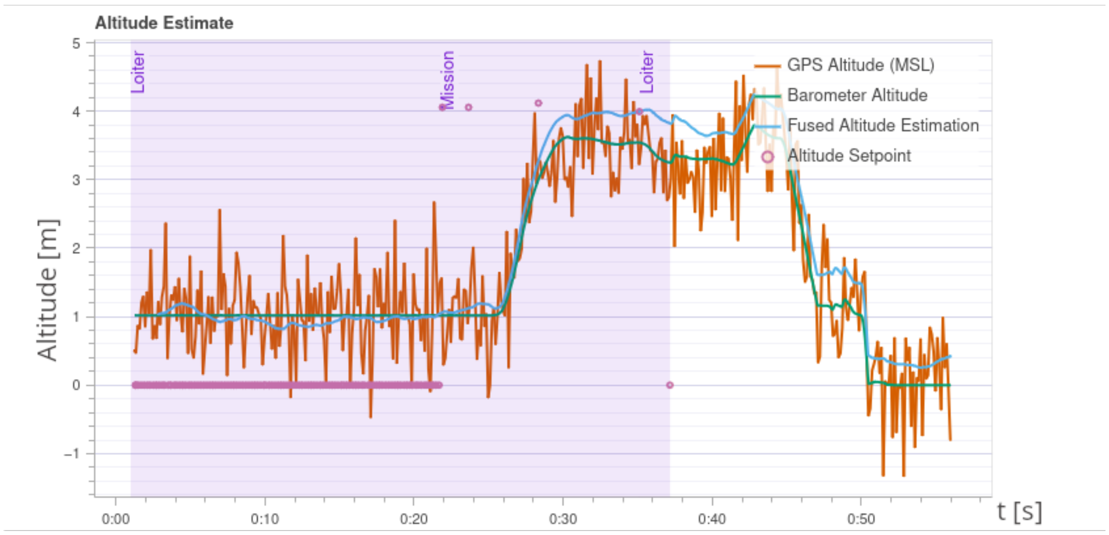


(a) Path Estimate with Default EKF2 Parameters.

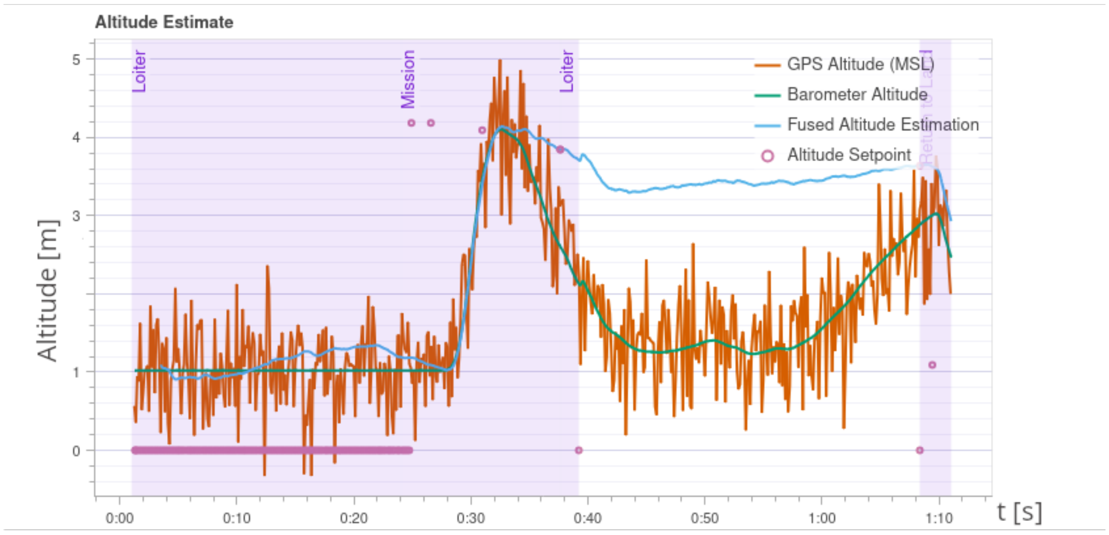


(b) Path Estimate with Modified EKF2 Parameters.

Figure 6.5: Comparison of Filter Performance in Two Scenarios.



(a) Altitude Estimate with Default EKF2 Parameters.



(b) Altitude Estimate with Modified EKF2 Parameters.

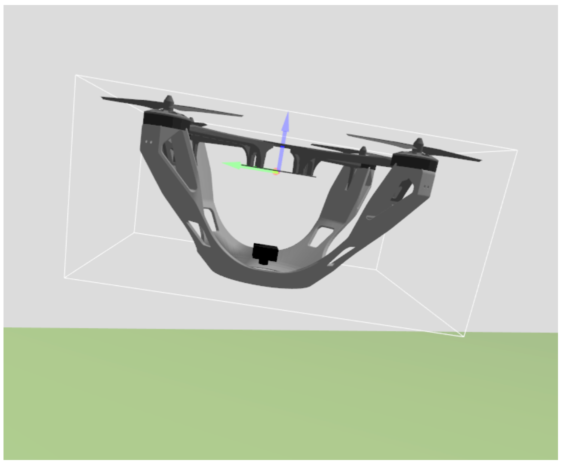
Figure 6.6: Comparison of Filter Performance in Two Scenarios.

6.4.4 Wind

The SDF file allows us to set the wind's linear velocity in the (x,y,z) directions. Thus, simulations were run for wind blowing in one, two, and three directions. For winds in one and two directions, the maximum wind speed that ensured docking was 3 m/s, but for wind with components in three directions, the maximum speed was 5 m/s. Wind was added to the world file of the simulation in the following way:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<sdf version="1.9">
  <world name="fixit">
      <wind>
      <linear_velocity>5 5 5</linear_velocity>
    </wind>
  </world>
</sdf>
```

Overall, the wind was a much harder variable to deal with when landing. The reason is that the force of the wind changes the direction of the resultant force acting on the drone, and thus, the drone must change the speed of its rotors to compensate for the wind and stay hovering. This change in speed causes the drone to tilt its axis with respect to the ground plane, which was the main source of error in the landing simulations for the following reason: the tilting of the drone's z-axis caused it to land on the cone with an improper orientation, which often produced unwanted collisions between the landing cone and the propellers.


(a) Tilting of the Drone's Frame.

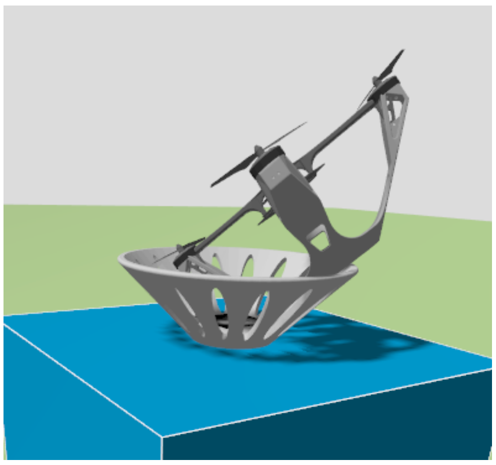

(b) Docking Fail.

Figure 6.7: Effect of Wind in Gazebo Simulation.

6.4.5 Distance Sensor for Height Measurements

As previously mentioned [4], structuring the code to meet a landing threshold with an estimate of the drone's altitude was not very effective since it required manually changing the threshold's value based on observations from the simulations. Therefore, it was decided to run simulations with a dedicated distance sensor that could provide a more accurate measurement of the drone's height.

The two sensors considered were UWB and lidar sensors. Since Gazebo does not include a plugin to include a UWB sensor, the first simulations were done with a lidar sensor built into PX4. The outcome of the simulations was that although the height measurement was more accurate, the added plugin made the simulation more computationally heavy, so much so that it ran at a factor of less than 30% of real-time. Therefore, ROS2 nodes that simulate the behavior of a UWB sensor were coded in Python so that an accurate altitude measurement could be taken without affecting the simulation's speed.

The UWB sensor, in its simplest form, is made up of a transmitter and a receiver called tag and anchor, respectively. The working principle of the sensor is:

- The tag polls the anchor by sending a radio wave at instant t_1.
- Once the anchor receives the poll, it sends a response signal after a fixed processing time t_{proc}.
- The tag receives the response at instant t_2 and computes the distance with the following equation $d = \frac{(t_2 - t_1) - t_{proc}}{2 \cdot c}$, where c is the speed of light.

This was implemented in Python by writing two nodes, one for the tag and another for the anchor. These nodes communicate with each other over two topics called */uwb_poll* and */uwb_respond*, which simulate the Poll/Response waves sent by the UWB sensor. The tag node then computes the altitude and publishes it to the */measured_altitude* topic at a rate of about 50 Hz.

Unlike a real UWB sensor, however, the polls and responses do not travel at the speed of light since they are ROS2 messages and not radio waves. Therefore, the tag node subscribes to the */odometry* topic to compute its current altitude with respect to its takeoff height. The OdometryPublisher plugin publishes the odometry topic, and it must be added to the drone's model.sdf file:

[4] See 6.4.3

```
<plugin name="gz::sim::systems::OdometryPublisher"
    filename="gz-sim-odometry-publisher-system">
  <tf_frame>odom</tf_frame>
  <odom_topic>/odometry</odom_topic>
  <child_frame>base_link</child_frame>
  <dimensions>3</dimensions>
</plugin>
```

The following graph shows a scheme of how all of the different ROS2 nodes used in the simulation interacted with each other:

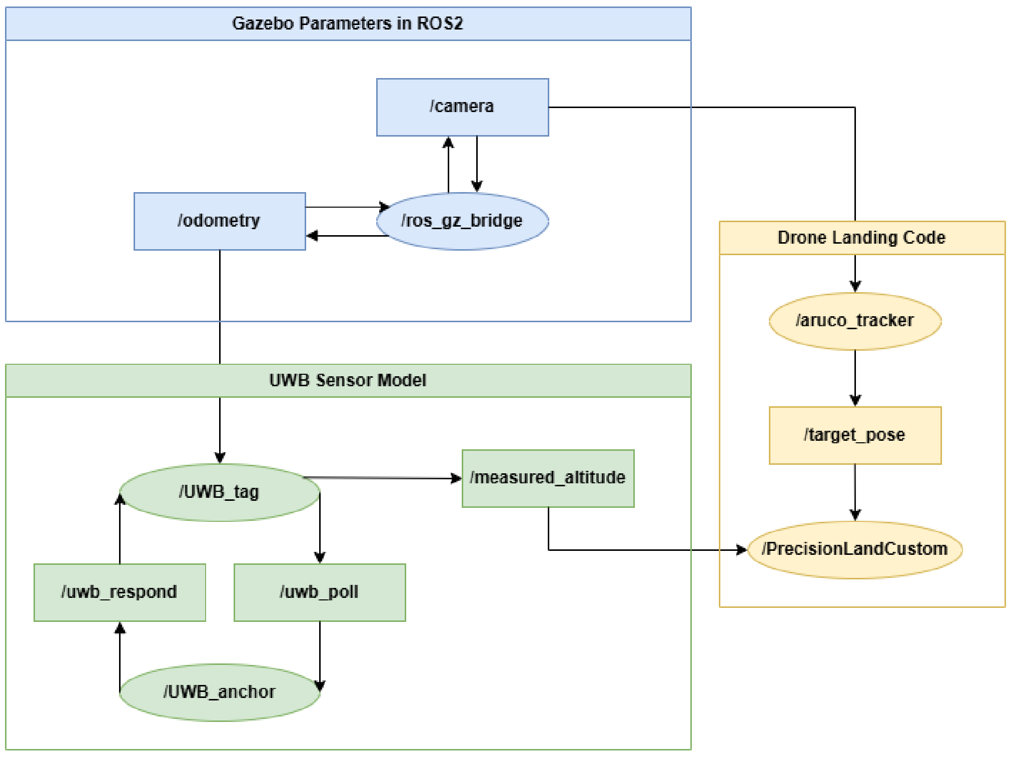


Figure 6.8: Graph with ROS2 Nodes and Topics.

Below, two tables are reported, in which the results of landing under different nonideal conditions are shown:

Sensor	Noise	Failure
Accelerometer	Static $\sigma_d = 0.01$	No
Accelerometer	Dynamic $\sigma_d = 0.005$	No
Gyroscope	Static $\sigma_d = 0.001$	No
Gyroscope	Dynamic $\sigma_d = 0.0005$	No

Table 6.4: Analysis of Docking Failure with Sensor Noise

Wind	Direction	Failure
≤ 3 m/s	X||Y	No
≥ 5 m/s	X||Y	Yes
≥ 6 m/s	X,Y	Yes
≥ 7 m/s	X,Y,Z	Yes

Table 6.5: Analysis of Docking Failure with Wind

Chapter 7

Conclusion and Future Work

7.1 Conclusion

This research aimed to develop a coupling system between a UAV and a UGV so the drone could land and recharge on the rover. A thorough state-of-the-art study of drone coupling and recharging systems was performed to review the different designs in academia and the market. Various solutions were analyzed, and the best tradeoff design was chosen. A prototype was modeled in SolidWorks, and the design was imported into Gazebo to test performance under non-ideal conditions. Finally, a state was coded onto the autonomous landing code [10] so that the drone would correct its altitude and land autonomously. A model of a UWB sensor was coded in Python to improve the altitude measurement in the descent phase.

It can be concluded that a simple design change of the cone's angle from 30° to 45° improved the 3D printing estimates by reducing the total printing time, the amount of printing supports, and the printing costs. The limitation of this design is that it adds extra weight to the drone's frame, which will affect its battery life. The total added mass, however, will be smaller than the mass found using the SolidWorks tool, since the 3D printing process allows the infill of the part to be modified.

The simulations' results showed that the proposed design efficiently corrected the drone's position in its descent phase, even in high sensor noise conditions. However, the design and the autonomous landing code had difficulties ensuring drone docking under strong winds.

7.2 Future Work

This thesis focused mostly on the mechanical coupling interface between the drone and the rover; therefore, the next step of the work would be to design an autonomous contact recharging mechanism. As mentioned in section 4.1, the board on the rover has a 24VDC output that can be used to recharge the drone's battery. The power would come from the DC converter and be fed to the drone through four semi-circular recharging electrodes, two of which would be placed on the cone and the other two on the landing gear. Since the drone uses a LiPo battery, however, additional electronic components would be required to properly manage the power delivery to the battery. The following is a non-exhaustive list of the possible components needed to automate the drone's recharging:

- Battery Charger IC to control the Constant Current/Constant Voltage charging cycles of the battery. Such as the BQ24773.
- Battery Management chip that ensures each cell reaches its maximum voltage. Such as the BQ76920.

Another consideration that would have to be further studied is the drone's orientation prior to landing, since this is something that must be corrected for the electrodes to come into full contact. The pose estimation made by the *aruco_tracker* node already estimates the marker's pose and sends this command to the drone when landing; however, a more robust analysis would have to be performed to ensure that the UAV properly aligns itself with the electrodes, perhaps by experimenting with different types of fiducial markers.

Bibliography

[1] DJI Dock 2. [URL].

[2] 3D CAD Model of the F450 Frame. [URL].

[3] Gazebo Logo. [URL].

[4] PX4-Autopilot Guide, . [URL].

[5] PX4 Logo, . [URL].

[6] QGC Logo. [URL].

[7] ROS2 Logo, . [URL].

[8] ROS-Gazebo Integration, . [URL].

[9] SDFormat. [URL].

[10] ARK Electronics. Tracktor Beam. [URL].

[11] T. Campi, S. Cruciani, M. Feliziani, and F. Maradei. High Efficiency and Lightweight Wireless Charging System for Drone Batteries. In *2017 AEIT International Annual Conference*, pages 1–6, 2017. doi: 10.23919/AEIT.2017.8240539.

[12] C. R. De Ceglia. Autonomous Precision Landing for UAVs. Master's Thesis, Politecnico di Torino, 2022.

[13] C. H. Choi, H. J. Jang, S. G. Lim, H. C. Lim, S. H. Cho, and I. Gaponov. Automatic wireless drone charging station creating essential environment for continuous drone operation. In *2016 International Conference on Control, Automation and Information Sciences (ICCAIS)*, pages 132–136, 2016. doi: 10.1109/ICCAIS.2016.7822448.

[14] CIM4.0. Landing FIXIT. [URL].

[15] F. Cocchioni, A. Mancini, and S. Longhi. Autonomous navigation, landing and recharge of a quadrotor using artificial vision. In *2014 International Conference on Unmanned Aircraft Systems (ICUAS)*, pages 418–429, 2014. doi: 10.1109/ICUAS.2014.6842282.

[16] Analog Devices. LTC2944 - 60V Battery Gas Gauge with Temperature, Voltage and Current Measurement. [URL].

[17] A. Kurs, A. Karalis, R. Moffatt, J. D. Joannopoulos, P. Fisher and M. Soljačić. Wireless Power Transfer via Strongly Coupled Magnetic Resonances. *Science*, 317(5834):83–86, 2007. doi: 10.1126/science.1143254. [URL].

[18] J. Guarino. HW and FW Description - v1.2, 2023.

[19] J. Guarino. Schematic FIXIT-M REV 1.1, 2023.

[20] Texas Instruments. BQ769x0 3-Series to 15-Series Cell Battery Monitor Family for Li-Ion and Phosphate Applications, 2022. [URL].

[21] A. Karalis, J.D. Joannopoulos and M. Soljačić. Efficient wireless non-radiative mid-range energy transfer. *Annals of Physics*, 323(1):34–48, 2008. ISSN 0003-4916. doi: https://doi.org/10.1016/j.aop.2007.04.017. [URL].

[22] D. Lee, J. Zhou, and W. T. Lin. Autonomous battery swapping system for quadcopter. In *2015 International Conference on Unmanned Aircraft Systems (ICUAS)*, pages 118–124, 2015. doi: 10.1109/ICUAS.2015.7152282.

[23] Steven Macenski, Tully Foote, Brian Gerkey, Chris Lalancette, and William Woodall. Robot operating system 2: Design, architecture, and uses in the wild. *Science Robotics*, 7(66):eabm6074, 2022. doi: 10.1126/scirobotics.abm6074. [URL].

[24] Microchip. ATmega2560 Datasheet. [URL].

[25] I. Munasinghe, A. Perera, and R. C. Deo. A Comprehensive Review of UAV-UGV Collaboration: Advancements and Challenges. *Journal of Sensor and Actuator Networks*, 13(6), 2024. ISSN 2224-2708. doi: 10.3390/jsan13060081. [URL].

[26] NASA. Perseverance's Selfie with Ingenuity, Apr 7, 2021. [URL].

[27] M. Peshkin and J. E. Colgate. Cobots. In *Industrial Robot, 25 (5)*, pages 355–341, 1999. [URL].

[28] Universal Robots. UR5e: Lightweight, versatile cobot. [URL].

[29] A. Saviolo, J. Mao, Roshan Balu T. M. B., V. Radhakrishnan, and G. Loianno. Autocharge: Autonomous Charging for Perpetual Quadrotor Missions. In *2023 IEEE International Conference on Robotics and Automation (ICRA)*, pages 5400–5406, 2023. doi: 10.1109/ICRA48891.2023.10161503.

[30] Skycatch, Inc. System and method for capturing aerial images. U.S. Patent US 9499265 B2, 2017. [URL].

[31] NASA Science Editorial Team. NASA's Ingenuity Helicopter to Begin New Demonstration Phase, Apr 30, 2021. [URL].

[32] WIZnet. W5500 Datasheet. [URL].

[33] J. Wu. Texas instruments - A Basic Guide to I2C. [URL].

Appendix A

Code

A.1 UWB Sensor Codes

A.1.1 uwb_tag.py

```python
#!/usr/bin/env python3
import rclpy
from rclpy.node import Node
from rclpy.qos import QoSProfile, ReliabilityPolicy
from std_msgs.msg import String, Float32
from nav_msgs.msg import Odometry

class UWBtag(Node):
    def __init__(self):
        super().__init__("UWB_tag_node") # Node's name
        self.get_logger().info("UWB Tag has started")
        self.resp = bool(False)
        self.first = bool(True)
        self.print = bool(True)
        self.drone_altitude = float

        self.z1 = None
        self.z2 = None

        # Define a QoS profile with BEST_EFFORT reliability to successfully
            subscribe to the topic
        qos = QoSProfile(
            reliability=ReliabilityPolicy.BEST_EFFORT,
            depth=10
        )

        # Define a Publisher:
        self.uwb_tag_pub = self.create_publisher(String,"uwb_poll",qos)
        self.altitude_pub =
            self.create_publisher(Float32,"measured_altitude",qos)
        timer_period = 0.02
        self.timer = self.create_timer(timer_period, self.timer_callback)
        self.i = 1

        # Define a Subscriber:
        self.uwb_tag_sub =
            self.create_subscription(String,"uwb_respond",self.uwb_tag_callback,qos)
        self.odometry_sub =
            self.create_subscription(Odometry,"/odometry",self.odometry_callback,qos)

    def timer_callback(self):
        msg=String()
        msg.data = 'poll ' + str(self.i)
        if(self.i==1):
            self.uwb_tag_pub.publish(msg)
            self.i+=1
        elif(self.resp):
            self.uwb_tag_pub.publish(msg)
            self.resp=False
            self.i+=1

    def uwb_tag_callback(self, msg):
        self.resp=True

        if(self.z2!=None and self.z1!=None):
            self.drone_altitude = self.z2-self.z1
            altitude_pub = Float32()
            altitude_pub.data = self.drone_altitude
            self.altitude_pub.publish(altitude_pub)

            if(self.i%100==0):
                self.get_logger().info('Altitude = ' + str(self.drone_altitude))

    def odometry_callback(self, msg):
        self.z2 = msg.pose.pose.position.z
        if(self.z1==None):
            self.z1 = self.z2

def main(args=None):
    rclpy.init(args=args)
    node = UWBtag()
    rclpy.spin(node)
    rclpy.shutdown()

if __name__ == '__main__':
    main()
```

A.1.2 uwb_anchor.py

```python
#!/usr/bin/env python3
import rclpy
from rclpy.node import Node
from rclpy.qos import QoSProfile, ReliabilityPolicy
from std_msgs.msg import String

class UWBanchor(Node):
    def __init__(self):
        super().__init__("UWB_anchor_node") # Node's name
        self.get_logger().info("UWB Anchor has started")
        self.poll = bool(False)

        # Define a QoS profile with BEST_EFFORT reliability to successfully subscribe to the topic
        qos = QoSProfile(
            reliability=ReliabilityPolicy.BEST_EFFORT,
            depth=10
        )
        # Define a Subscriber:
        self.uwb_anchor_sub = self.create_subscription(String,"uwb_poll",self.uwb_anchor_callback,qos)

        # Define a Publisher:
        self.uwb_anchor_pub = self.create_publisher(String,"uwb_respond",qos)
        self.j = 1
        timer_period = 0.02
        self.timer = self.create_timer(timer_period, self.timer_callback)

    def timer_callback(self):
        if(self.poll):
            msg=String()
            msg.data = 'resp ' + str(self.j)
            self.uwb_anchor_pub.publish(msg)
            self.j+=1
            self.poll = False

    def uwb_anchor_callback(self, msg):
        self.poll = True

def main(args=None):
    rclpy.init(args=args)
    node = UWBanchor()
    rclpy.spin(node)
    rclpy.shutdown()

if __name__ == '__main__':
    main()
```

A.2 Autonomous Landing

```
#include "PrecisionLand.hpp"
#include <px4_ros2/components/node_with_mode.hpp>
#include <px4_ros2/utils/geometry.hpp>
#include <Eigen/Core>
#include <Eigen/Geometry>

static const std::string kModeName = "PrecisionLandCustom";
static const bool kEnableDebugOutput = true;

using namespace px4_ros2::literals;

PrecisionLand::PrecisionLand(rclcpp::Node& node)
   : ModeBase(node, kModeName)
   , _node(node)
{
    ...
   _uwb_altitude_sub =
       _node.create_subscription<std_msgs::msg::Float32>("measured_altitude",
           rclcpp::QoS(1).best_effort(),
               std::bind(&PrecisionLand::uwbCallback, this,
               std::placeholders::_1));
    ...
}
void PrecisionLand::uwbCallback(const std_msgs::msg::Float32::SharedPtr msg)
{
   _drone_altitude = msg->data;
}

void PrecisionLand::updateSetpoint(float dt_s)
{
   case State::Search: {

      auto waypoint_position = _search_waypoints[0];
      _trajectory_setpoint->updatePosition(waypoint_position);

      float search_altitude = 3;

      if (positionReached(waypoint_position)){
         if (abs(_drone_altitude)>=search_altitude){
            _down = true;
            switchToState(State::ChangeAltitude);
            break;
         }else{
            _down = false;
            _z_search = _drone_altitude;
            switchToState(State::ChangeAltitude);
            break;
            }
        }
        break;
    }

    case State::ChangeAltitude: {
        _change_altitude = true;

        Eigen::Vector2f vel = calculateVelocitySetpointXY();

        if (_down){
            _trajectory_setpoint->update(Eigen::Vector3f(vel.x(), vel.y(),
                _param_descent_vel/2), std::nullopt,std::nullopt);

            if(abs(_drone_altitude)<=3 && !std::isnan(_tag.position.x())){
                RCLCPP_INFO(_node.get_logger(), "Altitude = %f\n",_drone_altitude);
                _approach_altitude = _vehicle_local_position->positionNed().z();
                switchToState(State::Approach);
                break;
            }
        }else{
            _trajectory_setpoint->update(Eigen::Vector3f(vel.x(), vel.y(),
                -1.f*_param_descent_vel/4), std::nullopt,std::nullopt);
            if(abs(_drone_altitude)>=(_z_search+0.5) &&
                !std::isnan(_tag.position.x())){
                RCLCPP_INFO(_node.get_logger(), "Altitude = %f\n",_drone_altitude);
                _approach_altitude = _vehicle_local_position->positionNed().z();
                switchToState(State::Approach);
                break;
            }
        }
        break;
    }
}
```