



A VISION BASED MULTIROTOR AIRCRAFT FOR USE IN THE SECURITY INDUSTRY

Masters Research Dissertation

For the qualification towards

MEng (Mechatronics)

Department of Mechatronics

Faculty of Engineering, The Built Environment and Information Technology

Benjamin David Nelson

Student number: 214006891

April 2020

Supervisor: Prof. Theo van Niekerk

Co-supervisor: Prof. Russell Phillips

Co-supervisor: Prof. Riaan Stopforth

Declaration

Benjamin David Nelson
30 Bibury Avenue
Linkside
Port Elizabeth
South Africa

I hereby solemnly declare that the dissertation submitted is a summary of my own research and that all sources used or referred to have been adequately referenced.

14 April 2020

A handwritten signature in black ink, appearing to read 'B. Nelson', written in a cursive style. The signature is positioned above a horizontal line.

B.D. Nelson

Acknowledgements

The following people and entities, who helped make this research possible, are hereby acknowledged:

- National Research Foundation, Eskom TESP programme, Department of Science and Technology (DST) ROSSA programme, MerSETA and Mr Karl du Preez, who provided the necessary funding for the research.
- Prof. Theo van Niekerk, for his encouragement to pursue a Master of Engineering, as well as supervise the research.
- Prof. Russell Phillips, who co-supervised the research and provided invaluable expertise in developing the system.
- Prof. Riaan Stopforth, who also co-supervised the research and provided insight into the use of new technology and software to integrate the system.
- Mr Paul Mooney, an expert in drones and UAVs, for overseeing the entire process of developing the system and providing constant guidance throughout the research.
- My parents, Mr Charles Nelson and Mrs Dianne Nelson, for their constant support and encouragement throughout my time at Nelson Mandela University.

Abstract

This research consisted of developing a vision based multirotor aircraft that could be used in the security industry. A second-hand aircraft was purchased and modified. The aircraft made use of a Pixhawk flight controller and a Odroid XU4 companion computer, which resulted in the computer injecting commands into the flight controller. Robot Operating System was installed and used on the companion computer to integrate the vision system and the aircraft. The vision system was designed to help develop a landing system where the aircraft would land on an ArUco marker. The vision system also allowed the aircraft to detect and follow humans. A Software in the Loop (SITL) was run alongside Gazebo, allowing the developed landing system and the human detecting system to be simulated and tested. The developed landing system was implemented on the aircraft, where the developed landing system was tested and compared to the aircraft's current GPS based landing system. The developed landing system obtained a better overall accuracy , while also taking longer to land the aircraft compared to the GPS based landing system. There were also numerous manual and autonomous test flights implemented on the aircraft.

Table of Contents

Declaration	I
Acknowledgements	II
Abstract	III
List of Figures	XII
List of Tables	XVII
Chapter 1: Introduction	1
1.1 Significance of Research.....	2
1.2 Aim	2
1.3 Objectives.....	3
1.4 Delimitation.....	3
1.5 Hypothesis.....	4
1.6 Dissertation Outline	4
1.7 Research Contributions	5
1.8 Publications	6
1.9 Potential Applications	6
1.10 Conclusion.....	7
Chapter 2: Background Research and Literature Review	8
2.1 Multirotor Aircraft Components.....	8
2.2 Arduino	8
2.3 Pixhawk Flight Controller.....	9
2.4 Autopilot Software	11
2.4.1 ArduPilot	11
2.4.2 PX4	11
2.5 Ground Control Station.....	11
2.6 MAVLink	12

2.7 Robot Operating System	15
2.8 Euler Angles & Quaternions	16
2.9 Attitude of an Aircraft	17
2.10 Vision	18
2.10.1 Object detection, recognition and tracking	18
2.10.2 Darknet and YOLO	18
2.11 Fiducial Markers	20
2.12 Current Implemented Solutions	21
2.12.1 The Sunflower System	21
2.12.2 Nightingale Security	22
2.13 Conclusion	23
Chapter 3: System Architecture	25
3.1 Multirotor Aircraft Dynamics	25
3.1.1 Drag	25
3.1.2 Thrust	26
3.1.3 Lift	26
3.1.4 Flying	28
3.1.5 Hovering	29
3.1.6 Increasing and Decreasing Altitude	29
3.2 Hardware Architecture	29
3.2.1 DJI vs Self-developed Aircraft	29
3.2.2 Frame	31
3.2.3 Flight Controller	32
3.2.3.1 Generic Pixhawk 1	32
3.2.3.2 Generic Pixhawk 2.4.6	33
3.2.3.3 Pixhawk 2 Cube	33

3.2.3.4 Original Pixhawk 1	34
3.2.4 Electronic Speed Controller	35
3.2.5 Brushless Motor & Propellers	35
3.2.6 Transmitter & Receiver	36
3.2.7 Battery	37
3.2.8 Flight Controller's Extra Components	37
3.2.9 Gimbal	38
3.2.10 Companion Computer	39
3.2.11 Camera	40
3.2.12 Extra Components	41
3.2.13 Additional Modifications	43
3.2.14 Designed Components	44
3.2.14.1 Gimbal Attachment v1	44
3.2.14.2 Gimbal Attachment v2	45
3.2.14.3 Leg Extension	46
3.2.14.4 GPS Case	47
3.2.14.5 Webcam Case	47
3.2.15 Final Hardware Components	48
3.2.16 Final Aircraft Hardware Architecture	49
3.2.17 Pixhawk Configuration	50
3.3 Software Architecture	52
3.3.1 Autopilot Software	52
3.3.2 Ground Control Station Software	53
3.3.3 PX4 Parameters	55
3.3.4 Companion Computer Software	55
3.3.4.1 Operating System	55

3.3.4.2 ROS.....	55
3.3.4.3 Terminator	56
3.3.5 Tarot Gimbal Software	56
3.3.6 Virtual Machine Software	57
3.3.7 Virtual Network Computing Software	58
3.4 Conclusion	59
Chapter 4: Vision System	60
4.1 Using Robot Operating System	60
4.2 Camera Calibration.....	62
4.3 ArUco Marker Detection	64
4.4 Human Detection	66
4.4.1 Using Darknet: YOLO with ROS	66
4.4.2 Calculating the ground distance.....	69
4.4.2.1 Calculating the distance of the aircraft from the bottom of the video frame	70
4.4.2.2 Calculating the distance of the person within the video frame	71
4.4.3 Centre a person in the frame	74
4.4.4 Developing coordinates for the aircraft using the calculated ground distance and calculated angle.....	76
4.5 Conclusion	80
Chapter 5: Integrated System	81
5.1 The Landing System.....	81
5.1.1 Reasoning behind developing the landing system.....	81
5.1.2 Basic overview of the landing system	82
5.1.3 Design	83
5.1.3.1 determinepose.cpp.....	84
5.1.3.2 flyaircraft.cpp	90

5.1.3.3 talker_autoland.cpp	91
5.1.3.4 talker_coordinates.cpp	91
5.1.3.5 talker_descent.cpp	92
5.1.3.6 talker_velocity.cpp	92
5.1.3.7 talker_flightmode.cpp	92
5.1.3.8 talker_quat.cpp	93
5.1.3.9 listener_status.cpp	93
5.1.4 Landing System Flow Chart.....	93
5.2 Human Detection System.....	93
5.2.1 Basic overview of the human detection system	94
5.2.2 Design	94
5.2.2.1 determinepose.cpp	95
5.2.2.2 boxinfo.cpp	96
5.2.2.3 boundingboxmove.cpp	97
5.2.2.4 EulerToQuat.py	97
5.2.2.5 QuatToEuler.py	98
5.2.3 Human detection system flow chart.....	99
5.3 User Requirements Specification	99
5.4 Conclusion.....	99
Chapter 6: Simulation	100
6.1 Setup of the Simulation	100
6.2 Landing System Simulation	102
6.3 Human Detection System Simulation	106
6.4 Conclusion.....	110
Chapter 7: Testing and Discussion	111
7.1 Test Flights	112

7.1.1 Manual Test Flight	112
7.1.2 Autonomous test flight performing a mission	113
7.1.3 An autonomous test flight where a mission is injected into the flight controller by a separate system	114
7.1.4 An autonomous test flight using ROS	116
7.1.5 The aircraft autonomously landing using ROS	116
7.2 Human Detection System.....	124
7.2.1 Comparing Darknet’s YOLO version 3 to Darknet’s YOLO Tiny Version 3	125
7.2.2 Differentiating Between a Human and Another Object.....	127
7.2.3 Correcting the Calculated Distance	128
7.3 Improvements to the System.....	134
7.4 Conclusion.....	135
Chapter 8: Conclusion.....	136
8.1 Conclusion.....	136
8.2 Recommendations for Future Research.....	138
Bibliography	140
Appendices	149
Appendix 1.1: South Africa’s Crime Statistics	149
Appendix 2.1: Darknet’s YOLO Dataset	150
Appendix 3.1: Gimbal Attachment v1 CAD Drawing	151
Appendix 3.2: Gimbal Attachment v2 CAD Drawing	152
Appendix 3.3: Leg Extension CAD Drawing.....	153
Appendix 3.4: GPS Case CAD Drawings	154
Appendix 3.5: Webcam CAD Drawings.....	156
Appendix 3.6: Pixhawk Schematic and Pinout.....	159
Appendix 3.7: Pixhawk PX4 Parameters.....	163

Appendix 4.1: Checkerboard Used for Camera Calibration	172
Appendix 5.1: determinepose.cpp node	173
Appendix 5.2: flyaircraft.cpp node	195
Appendix 5.3: talker_autoland.cpp node	202
Appendix 5.4: talker_coordinates.cpp node	204
Appendix 5.5: talker_descent.cpp node	206
Appendix 5.6: talker_velocity.cpp node	208
Appendix 5.7: talker_flightmode.cpp node	210
Appendix 5.8: talker_quat.cpp node	212
Appendix 5.9: listener_status.cpp node.....	214
Appendix 5.10: boxinfo.cpp node	215
Appendix 5.11: boundingboxmove.cpp node	220
Appendix 5.12: EulerToQuat.py node	226
Appendix 5.13: QuatToEuler.py node	228
Appendix 5.14: Quaternion Simulator.....	230
Appendix 5.15: Developed Landing System Flow Chart	232
Appendix 5.16: Human Detection System Flow Chart	233
Appendix 6.1: Terminals Used for the Developed Landing System	234
Appendix 6.2: Additional Terminals Used for the Human Detection System.....	236
Appendix 6.3: Testing of the Developed Landing System in Gazebo	237
Appendix 6.4: Nodes Tree for the Developed Landing System.....	240
Appendix 6.5: Nodes Tree for the Human Detection System.....	241
Appendix 7.1: Arduino PWM Script	242
Appendix 7.2: InsertWaypoints.cpp node	248
Appendix 7.3: Testing of the Developed Landing System on the Actual Aircraft	253

Appendix 7.4: Tables for Test 1, Test 2 and Test 3 for the Human Detection System..... 256

List of Figures

Figure 2.1: An Arduino Uno board (Sparkfun, 2015).....	9
Figure 2.2: Pixhawk 1 flight controller (PX4, 2019b).....	10
Figure 2.3: MAVLink v1.0 packet representation (Koubaa, et al., 2015).....	13
Figure 2.4: Rotations about the aircraft's current frame (CHRobotics, 2019b)	18
Figure 2.5: YOLO being used to detect multiple objects within a frame	19
Figure 2.6: Examples of fiducial markers (Mostashiri & Dhupia, 2018)	20
Figure 2.7: The Sunflower System (Sunflower Labs Inc., 2019).....	22
Figure 2.8: Nightingale Security system (ISC West, 2019).....	23
Figure 3.1: Forces acting on a multirotor aircraft	26
Figure 3.2: Coanda Effect (Aviation History, 2015).....	27
Figure 3.3: The forces applied to an aircraft (NASA, 2015)	27
Figure 3.4: Direction of rotation for the brushless motors (ArduPilot, 2019)	28
Figure 3.5: DJI F550 Flame Wheel (Build Your Own Drone, 2019)	32
Figure 3.6: Generic Pixhawk 2.4.6 (Banggood, 2019)	33
Figure 3.7: (a) Pixhawk 2 Cube (GetFPV, 2019) (b) Here+ GPS module (Heli Engadin, 2019).....	34
Figure 3.8: Afro-ESC 30A ESC (RC Groups, 2013).....	35
Figure 3.9: (a) Prop Drive brushless motor (HobbyKing, 2019) (b) APC 10 x 4.7 inches propeller (Hobby-Miracle, 2019).....	36
Figure 3.10: (a) FrSky Taranis Q X7 transmitter (GetFPV, 2019) (b) FrSky X8R receiver (PorcupineRC, 2019).....	36
Figure 3.11: Gens ace 3300mAh LiPo battery (Unmanned Tech, 2019).....	37
Figure 3.12: (a) GPS Module and GPS Stand (Amazon, 2019) (b) 433Mhz telemetry (Readytosky, 2019).....	37
Figure 3.13: Tarot ZYX T-3D V gimbal (Tarot, 2017).....	38
Figure 3.14: Tarot T4-3D Gimbal	39

Figure 3.15: (a) Odroid XU4 (Hardkernel, 2019a) (b) Comparing the read and write speeds (Hardkernel, 2019b).....	40
Figure 3.16: Logitech C920 webcam (Logitech, 2019)	40
Figure 3.17: Unitek USB3.0 4-Port Hub (Unitek, 2019)	41
Figure 3.18: Hobbywing UBEC 8A (Hobbywing, 2015).....	41
Figure 3.19: FTDI cable (left) and Wi-Fi module (right).....	42
Figure 3.20: Modified D-Link DSL-2750U modem router.....	43
Figure 3.21: Polystyrene balls used to help with the direction of the aircraft.	43
Figure 3.22: Modification of keeping the tail polystyrene ball and supporting the rod	44
Figure 3.23: Part designed to attach the gimbal to the bottom plate of the aircraft's fuselage.....	45
Figure 3.24: New part designed to attach the gimbal to the aircraft's bottom plate. .	46
Figure 3.25: Part designed to extend the legs on the aircraft.	46
Figure 3.26: (a) GPS case designed in CAD (b) GPS case opened.....	47
Figure 3.27: (a) the front of the webcam case (b) the back of the webcam case, which is fastened to the Tarot T4-3D gimbal.....	48
Figure 3.28: Block diagram for the final hardware architecture of the aircraft.....	49
Figure 3.29: Peripherals connected to the Pixhawk.....	51
Figure 3.30: Receiver, gimbal and six ESCs connected to the Pixhawk.....	52
Figure 3.31 Terminator used to group terminals	56
Figure 3.32: Tarot Gimbal Software	57
Figure 3.33: VNC Viewer running on a MacBook Pro laptop	58
Figure 4.1: Camera calibration being performed using the checkerboard (Shipitko, 2017).....	63
Figure 4.2: ArUco marker with an ID of 7.....	65
Figure 4.3: Person detected on a field, where a bounding box was estimated.....	68

Figure 4.4: Diagram demonstrating the frame seen by the camera	69
Figure 4.5: Angle between the aircraft and the bottom of the camera's field of view	70
Figure 4.6: Angle between the aircraft and the top of the camera's field of view.....	71
Figure 4.7: Angles within the cameras field of view	72
Figure 4.8: Triangle formed within the camera's field of view	72
Figure 4.9: Same triangle as in figure	73
Figure 4.10: Smaller triangle formed underneath the bounding box of the person ...	73
Figure 4.11: Top view of the aircraft detecting a person	75
Figure 4.12: Frame of reference for ROS and PX4, where North East Down (NED) on the left and East North Up (ENU) on the right (PX4, 2019h).....	77
Figure 4.13: Example to demonstrate how the co-ordinate is calculated.....	77
Figure 4.14: Angles around the aircraft used to calculate the new co-ordinate	78
Figure 4.15: Quadrants around the aircraft	79
Figure 5.1: Code used to implement the quaternion multiplication (Euclidean Space, 2017).....	89
Figure 6.1: Quadcopter model developed by PX4 for Gazebo	101
Figure 6.2: The Gazebo setup for the landing system.	102
Figure 6.3: The aircraft has taken off, where the window on the right shows the camera feed.	103
Figure 6.4: Camera feed on when the aircraft had correctly aligned itself with the ArUco marker	104
Figure 6.5: (a) Case 1 (b) Case 2	104
Figure 6.6: (a) Case 3 (b) Case 4	105
Figure 6.7: (a) Video of a person walking that was published by the video_stream_opencv package (b) bounding box that was formed using the darknet_ros package.....	106
Figure 6.8: The Gazebo setup for the human detection system	107

Figure 6.9: Case 1, where (a) is the orientation of the aircraft before the rotation and (b) is the orientation of the aircraft after the rotation	108
Figure 6.10: Case 1, where (a) is the pose of the aircraft before the repositioning and (b) is the pose of the aircraft after the repositioning.....	108
Figure 6.11: Case 2, where (a) is the orientation of the aircraft before the rotation and (b) is the orientation of the aircraft after the rotation	109
Figure 6.12: Case 2, where (a) is the pose of the aircraft before the repositioning and (b) is the pose of the aircraft after the repositioning.....	109
Figure 7.1: The multirotor aircraft flying at PERF	112
Figure 7.2: The mission setup for PERF that was done on QGroundControl	113
Figure 7.3: The mission performed that was seen on QGroundControl.....	114
Figure 7.4 The Arduino injection setup	115
Figure 7.5: MAVROS terminal showing three new waypoints injected into the flight controller	116
Figure 7.6: The aircraft and the ArUco marker before take off.....	117
Figure 7.7: The multirotor aircraft hovering above the ArUco marker when a person held the marker	118
Figure 7.8: (a) aircraft hovering above the marker (b) aircraft disarming, resulting in the aircraft dropping out of the air	119
Figure 7.9 The plot for the accuracy of landing the aircraft for both landing systems	123
Figure 7.10: The plot for the time taken to land the aircraft for both landing systems	123
Figure 7.11: The detected person using Darknet's YOLO version 3.....	125
Figure 7.12: The detected person using Darknet's YOLO tiny version 3	126
Figure 7.13: Darknet's YOLO tiny version 3 incorrectly identified the person as a bird	127
Figure 7.14: A person and a dog was detected by the vision system.....	127

Figure 7.15: The bounding box information of the person in their original position.	128
Figure 7.16: The bounding box information of the person after they had walked a few steps to their left.....	129
Figure 7.17: The bounding box information of the person after the DJI had rotated to centre the person in the video frame.....	129
Figure 7.18: Graph plotted and equation developed for Test 1's data	132
Figure 7.19: Graph plotted and equation developed for Test 2's data	132
Figure 7.20: Graph plotted and equation developed for Test 3's data	133

List of Tables

Table 2.1: Description of each byte in MAVLink v1.0 protocol (MAVLink, 2019).....	14
Table 2.2: Terms frequently used when using ROS	16
Table 3.1: Advantages and disadvantages of using a DJI Phantom 4 Advanced.....	30
Table 3.2: Advantages and disadvantages of developing an aircraft.....	30
Table 3.3: List of components that can be found on the final design of the aircraft..	48
Table 3.4: List of other components not directly used on the aircraft.....	49
Table 3.5: Peripherals connected to the Pixhawk	50
Table 3.6: Colour code of the wiring of the FTDI cable and their pinouts	51
Table 3.7: Description of the flight modes used on the aircraft while using ArduPilot firmware (ArduPilot, 2019)	54
Table 3.8: Description of the flight modes used on the aircraft while using PX4 firmware (PX4, 2019c)	54
Table 4.1: Using OpenCV for the vision system	60
Table 4.2: Using ROS to develop the entire system	61
Table 4.3: ROS packages used in the build.....	62
Table 4.4: Frame of reference for ROS and PX4.....	76
Table 5.1: The topics subscribed to in the determinepose.cpp node.....	84
Table 5.2: The topics published to in the determinepose.cpp node.....	85
Table 5.3: Quaternion formed if the aircraft needs to rotate clockwise ($0^\circ < \theta \leq 90^\circ$)	87
Table 5.4: Quaternion formed if the aircraft needs to rotate clockwise ($90^\circ < \theta < 180^\circ$)	87
Table 5.5: Quaternion formed if the aircraft needs to rotate counter clockwise ($90^\circ < \theta \leq 180^\circ$).....	88
Table 5.6: Quaternion formed if the aircraft needs to rotate counter clockwise ($0^\circ \leq \theta \leq 90^\circ$).....	88

Table 5.7: The topics subscribed to in the flyaircraft.cpp node	90
Table 5.8: The topics published to in the flyaircraft.cpp node	91
Table 5.9: The extra topic subscribed to in the determinepose.cpp node.....	95
Table 5.10: The topic subscribed to in the boxinfo.cpp node.....	96
Table 5.11: The topics published to in the boxinfo.cpp node	96
Table 5.12: The topics subscribed to in the boundingboxmove.cpp node	97
Table 5.13: The topic published to in the boundingboxmove.cpp node	97
Table 5.14: The topic subscribed to in the EulerToQuat.py node	98
Table 5.15: The topic published to in the EulerToQuat.py node	98
Table 5.16: The topic subscribed to in the QuatToEuler.py node	98
Table 5.17: The topic published to in the QuatToEuler.py node	98
Table 7.1: Type of data recorded for each landing	121
Table 7.2: The initial heading of the aircraft recorded before testing each landing method	121
Table 7.3: Data recorded for the landing of the aircraft by using the GPS module.	122
Table 7.4: Data recorded for the landing of the aircraft by using the developed landing system.....	122
Table 7.5: Test 1 points calculated	131
Table 7.6: Test 2 points calculated	132
Table 7.7: Test 3 points calculated	133

Chapter 1: Introduction

In today's modern era, there has been an increase in the number of people flying aircraft. Specifically, there has been an increase in the number of people flying drones. A drone, which can be formally classified as an unmanned aerial vehicle (UAV), is a multirotor aircraft that is lifted and propelled using multiple rotors (Parker, 2018). A drone does not have a pilot sitting inside it, which means it is operated either by software or a remote pilot (Parker, 2018). This type of aircraft varies in multiples sizes, depending on the number of motors and legs that are attached and installed onto the centre frame of the aircraft. Some examples are a tricopter (3 legs & motors), quadcopter (four legs & motors) and a hexacopter (6 legs & motors).

To this day, there is still uncertainty whether a drone can be classified as a UAV. This is due to the fact that a UAV generally refers to any military aircraft that operates without a pilot and can be reused (Parker, 2018). Throughout this research, the terms UAV and multirotor aircraft will be used.

The development of multirotor aircraft has significantly helped humans accomplish tasks that never seemed achievable before. Some examples are: use in military operations for combat and reconnaissance purposes (Quadcopter Arena, 2017); use by law enforcement agencies in search and rescue operations (Quadcopter Arena, 2017); great research opportunities for universities (e.g. robotics, flight control and real-life systems) (Quadcopter Arena, 2017); and use for recreational purposes, such as flying as a hobby or competitive racing (Quadcopter Arena, 2017). Another big userbase of multirotor aircrafts are photographers who shoot photos and videos from difficult positions, resulting in their content looking stunning and memorable. Companies such as DJI have developed such aircraft which contain a three-axis gimbal attached beneath the frame of the aircraft. A GoPro is even attached to these gimbals to capture the moment. A gimbal is regarded as a mechanism that keeps an instrument (e.g. camera) at a set orientation on an aircraft or ship. This will reduce the shakiness of the video footage captured. (UAV Systems International, 2018). This development of attaching a camera to a multirotor aircraft has increased the scope of research in many different fields.

1.1 Significance of Research

In 2017, the Institute for Economics and Peace (IEP) conducted a study rating the level of peace for a number of countries around the world. This study was called the Global Peace Index, where it was performed in the year 2017. The Institute rated South Africa to be the 123rd most peaceful country in the world, out of a total of 163 countries and districts that were measured (Business Tech, 2017). This indicates that South Africa is one of the most unsafe countries in the world. The document also revealed the high crime rate in our country, where violent crime and homicide were both rated 5 out of 5 (Institute for Economics & Peace, 2017).

These levels of violence and insecurity have a massive impact on the country's economy. The IEP measured the cost of violence in South Africa at 22.3% of GDP, which is the equivalent of R1.92 trillion (Business Tech, 2017).

For a number of years, South Africa has been heavily affected by crime. Despite the development of technology in today's world, the country still seems to suffer from high crime statistics. This is probably due to the increase in unemployment in the country. The South African Police Service (SAPS) annually collects and releases the national crime statistics for a given year. Appendix 1.1 reflects the extent of crime taking place between the years of 2018 and 2019.

These statistics reveal that South Africa is in need of finding a new way to reduce crime in the country. To help with this, the idea is to perform research that would consist of developing an aircraft that could in the future help reduce the overall crime rate in the country.

1.2 Aim

The aim of this research is to add to the development of an autonomous multicopter aircraft that could be used in the security industry by making use of a vision system. The aim includes using the vision system to help with the landing system of the aircraft as well as guide and direct the aircraft to new positional coordinates based on detecting human interaction.

1.3 Objectives

The objectives of this dissertation consist of the following:

1. The performance of a literature review that could help understand the following main topics:
 - a. Open source flight controllers
 - b. ROS
 - c. Object detection, object recognition and object tracking
 - d. Fiducial markers
2. The development of a vision based multirotor aircraft for use in the security industry
 - a. Establishing a hardware architecture to define the interaction between all the components used, whilst still allowing the system to be compact and modular.
 - b. Establishing a software architecture to define all the software used as well as to demonstrate how all the programs developed will interact with one another.
3. The development of an autonomous multirotor aircraft
 - a. The development of a multirotor aircraft can be demonstrated.
 - b. The aircraft can perform a fully autonomous mission, where the flight route is predefined.
 - c. The aircraft can perform a fully autonomous mission, where new coordinates for the flight route can be injected into the flight controller by other methods. This injection of new coordinates can take place during flight.
4. The development of a vision system
 - a. Develop an algorithm that can be used to identify a marker to help guide the aircraft during its landing phase.
 - b. Develop an algorithm that can be used to identify human interaction as well as produce a new positional coordinate for the aircraft to fly towards.

1.4 Delimitation

Due to vast number of subsystems in this research, all being integrated into a single system, there were a few limitations that occurred within the integrated subsystems:

- Due to the limited funding available, superior components would not be able to be purchased. Their performances would limit the accuracy and reliability of the system.
- Due to the small space available on the aircraft, the components used would need to be relatively small (e.g. the onboard computer). This would limit the processing ability of the system.
- The testing of the system is heavily dependent on the weather, limiting when the system can be tested.
- The testing of the system is limited to the aircraft's battery capacity.

1.5 Hypothesis

A software and hardware architecture and a supporting digital simulation environment can be designed to construct a vision based multicopter aircraft for the security industry, where the focus will be on developing an alternative landing system and human detection system.

1.6 Dissertation Outline

This dissertation will consist of the following chapters:

1. Chapter 1: Introduction

This chapter will provide an introduction to this research, where the aim, objectives, significance, delimitation, research hypothesis, research contributions and publications will all be discussed.

2. Chapter 2: Background Research and Literature Review

This chapter will consist of multiple sections that will bring relevant background to this research.

3. Chapter 3: System Architecture

This chapter will discuss the components used, the designed components that were developed and finally the hardware and software architecture.

4. Chapter 4: Vision System

This chapter will describe the vision system that was established to help with the landing system as well as for detecting humans.

5. Chapter 5: Integrated System

This chapter will discuss how ROS is used to integrate the system as well as explain how the vision system and the aircraft is integrated with ROS, allowing for a complete system.

6. Chapter 6: Simulation

This chapter will discuss all the simulations that were performed in testing the landing system and the human detection system.

7. Chapter 7: Testing and Discussion

This chapter will discuss the test flights that were performed. These test flights will consist of:

- A manual test flight.
- An autonomous test flight performing a mission.
- An autonomous test flight where a mission is injected into the flight controller by a separate system.
- An autonomous test flight using ROS.
- The aircraft autonomously landing using ROS.

The chapter will include a data analysis on the ROS developed landing system, where the GPS landing system will be compared to the ROS developed landing system. A discussion will be performed for all the sections, including the human detection system. Improvements to the system will also be mentioned.

8. Chapter 8: Conclusion

This chapter will conclude the dissertation, suggesting whether the hypothesis, aim and objectives were achieved; future development to the system; and whether the research made any significant contribution to the field of engineering.

1.7 Research Contributions

After completing this research, the following research contributions will be obtained:

- The development of a vision system that can be used to detect ArUco markers and detect objects within a video frame.
- The development of a multirotor aircraft that can be flown manually or be used to perform autonomous missions.
- The development of a landing system that makes use of a vision system to land an aircraft on an ArUco marker, providing a more accurate system than the

GPS based landing system as well as a cheaper alternative system to using a RTK system.

- The development of a human detection system that makes use of a vision system to command an aircraft to reposition and re-orientate itself to follow a moving person.
- A vision system and multirotor aircraft that is integrated using ROS.

1.8 Publications

A conference paper, called “Autonomous Landing of a Multirotor Aircraft on a Docking Station”, was written together with Jacques du Preez, a fellow Mechatronics student, pursuing a Master of Engineering. The paper was submitted and accepted for the RobMech conference, which took place in Cape Town in January 2020. Jacques attended the conference, where he presented a poster, demonstrating the work of both writers.

1.9 Potential Applications

One potential application for the developed system would be to use it as part of a security surveillance system. For instance, if a home alarm system were to be triggered by an unwanted intruder, a signal could be sent to the developed system, requesting the multirotor aircraft to fly autonomously to the specific location. Once the aircraft arrived at the house concerned, it would then survey the property, searching for any intruders by making use of the onboard vision system. If an intruder were to be detected, the aircraft would send a signal to the security company’s control room, requesting permission to follow him/her. If the necessary consent were received, the aircraft would use its vision system to follow the intruder. The vision system would constantly instruct the aircraft to adjust its position and orientation to centre the intruder in the middle of the video feed of the vision system, as well as to keep the aircraft a fixed distance away from him/her. When the aircraft is not being used, it could make use of a docking station to charge the batteries of the aircraft as well as to protect the aircraft from any harmful weather.

The above idea was presented to Atlas Security, a security company based in Port Elizabeth, South Africa.

1.10 Conclusion

This chapter provided an overall introduction to the research, where the research was defined to be the development of a vision based multirotor aircraft that can be used in the security industry. The aim of the research was provided along with a list of objectives to be achieved. The significance of the research was discussed, shedding light on the current crime rates in South Africa. The delimitation and the hypothesis for the research were also discussed. A layout for the dissertation was provided, mentioning what will be discussed in each chapter. The research contributions were outlined as well as mentioning the publication of a conference paper. The following chapter, which is the background research and literature review, will discuss all the relevant background material that was utilised throughout the research.

Chapter 2: Background Research and Literature Review

The background research and literature review for this research will be approached in the following manner: Firstly, the components required to fly a multirotor aircraft will be discussed; secondly, topics that are related to the aircraft will be researched, such as an Arduino, Pixhawk flight controllers, autopilot software, ground control stations, MAVLink, ROS, Euler angles and quaternions, Darknet: YOLO and fiducial markers; thirdly, the crime rates in South Africa will be investigated; and finally the current implemented security solutions consisting of multirotor aircraft will be discussed.

2.1 Multirotor Aircraft Components

The main components that are commonly found on a multirotor aircraft as well as a basic description of the component can be seen below (Kadamatt, 2017):

1. Frame: It is the body of the aircraft. It is responsible for holding all the components of the aircraft.
2. Brushless Motors and Propellers: The brushless motors and propellers work together to produce the thrust for the aircraft.
3. Electronic Speed Controllers (ESC): They are responsible for controlling the rate at which the brushless motors spin depending on the pulse width modulation (PWM) that each ESC has received from the flight controller.
4. Flight Controller: It is the “brains” of the aircraft, where it is constantly making necessary calculations and producing PWM signals for the ESCs to control the motors.
5. Transmitter and Receiver: They are used to control the aircraft. A pilot will make use of the transmitter to fly the aircraft, whereas the receiver will receive the command performed on the transmitter and pass it through to the flight controller.
6. Battery: This is used to provide power to the aircraft.

2.2 Arduino

An Arduino is an open source physical computing platform used for creating interactive objects that can stand alone or collaborate with software on a computer. Arduino was designed for artists, designers and others who want to incorporate physical computing

into their designs, without first having to become electrical engineers (Banzi & Shiloh, 2014).

The Arduino boards are able to read inputs (e.g. light on a sensor, pressing a button) and turn it into an output (e.g. activating a motor, turning on an LED) (Banzi & Shiloh, 2014). It is possible to instruct the Arduino board by sending a set of instructions to the microcontroller that is fixed on the board. This is done by using software, known as the Integrated Development Environment (IDE), which is freely available off the Arduino website. The language, used in the IDE to program the microcontroller, consists of a set of C/C++ functions (Arduino, 2017).

There are a number of different versions of the Arduino boards available (e.g. Due, Mega, Micro, Uno, etc.) (Sparkfun, 2015). Each model has its own specifications, with different voltage inputs, different microcontrollers, etc.

An example of an Arduino board may be seen in Figure 2.1.



Figure 2.1: An Arduino Uno board (Sparkfun, 2015)

2.3 Pixhawk Flight Controller

Pixhawk is an independent open-hardware project that aims to provide the standard for readily-available, high-quality and low-cost autopilot hardware designs, which can

be used in the academic, hobby and developer communities (Pixhawk, 2019). The Pixhawk project was first started at ETH Zurich as both an open-source hardware and software project to create a flight controller (LambDrive, 2016). In the first designs, the Pixhawk started out as two separate boards, one called the PX4FMUv1 and the other called PX4IOv1. The PX4FMUv1 was the flight measurement unit (FMU) and the PX4IOv1 was the inputs and outputs. Eventually, these two boards were combined into one, which was called the Pixhawk 1 (LambDrive, 2016). The Pixhawk 1 can be found in Figure 2.2.



Figure 2.2: Pixhawk 1 flight controller (PX4, 2019b)

At the time, 3D Robotics was chosen as the prominent manufacturer. However, since it is an open-hardware project, the schematics and PCB design files are freely available for anyone to use, modify and manufacture themselves. Hence, there are many clone Pixhawk flight controllers available to be purchased from Chinese websites. The beauty of its being open-source is that it has allowed for many projects to be developed as well as be freely available to incorporate into other projects. The Pixhawk does not just have to be used as a flight controller, but has the ability to be used within a ground based project. An example of where a Pixhawk can be used on a ground based project is where the board is used on a four-wheeled rover, enabling it to travel along a terrain.

2.4 Autopilot Software

The autopilot software is the software that is installed onto the Pixhawk. Since the Pixhawk is an open-source project, its purpose is to be used with an open-source autopilot software. There are a number of open-source autopilot software available, however two of the main ones are ArduPilot (APM) and PX4.

2.4.1 ArduPilot

ArduPilot, often referred to as APM, is one of the most advanced, full featured and reliable open-source autopilot software available. It has been developed for over 5 years by a team of professional engineers and computer scientists. The software is capable of controlling a numerous vehicles, such as:

1. Airplanes
2. Multirotors
3. Helicopters
4. Boats
5. Submarines

New software for aircraft, such as quad-planes and compound helicopters, are currently being developed. The software has been installed on over 1 000 000 vehicles world-wide due to the software being one of the most tested and proven autopilot software available (ArduPilot, 2016).

2.4.2 PX4

PX4 is an open-source flight control software that has been designed for drones and other unmanned vehicles. The project provides a flexible set of tools for drone developers to share technologies that create tailored solutions for specific drone applications. The software was developed alongside the original Pixhawk project at ETH Zurich. Today, the project consists of more than 300 global contributors and is used by some of the world's most-innovative companies across a wide range of drone industry applications (PX4, 2018).

2.5 Ground Control Station

A ground control station (GCS) is generally referring to a software application, which utilizes a ground-based computer, that is able to communicate with a UAV using a

wireless telemetry. The GCS is able to display real-time data of the UAV's performance, position and orientation, which includes displaying typical instruments that you would likely find in a real aeroplane. The GCS is also able to perform other necessary tasks, (e.g. setting up a mission for an aircraft, sending the aircraft on a mission, etc.) (ArduPilot, 2019). Depending on which GCS software is being used, some offer the ability to load the aircraft's firmware onto its flight controller, as well as set up the aircraft so that it may be correctly used for flying.

Due to the vast number of open-source onboard software for UAVs, this has led to a number of open-source GCS software available on the internet. Some examples are:

- MAVProxy
- Mission Planner
- QGroundControl (QGC)

The GCS that will be used for this research will be QGroundControl due to its ease of use as well as support for multiple types of UAVs. The software is available on multiple platforms, which are:

- Windows
- macOS X
- Linux
- Android
- iOS

This will allow for the aircraft to be quickly and easily adjusted on portable devices (e.g. cell phone, tablet, etc.) when testing is being performed on an open field.

Most GCS communicate with a UAV using MAVLink protocol. More information on this can be found in section 2.6.

2.6 MAVLink

Micro Air Vehicle Link, which is commonly referred to as MAVLink, is a lightweight messaging protocol used for communicating with small unmanned vehicles (MAVLink, 2019). MAVLink, which was first released in early 2009 by Lorenz Meier, is a reliable communication protocol as it provides methods for detecting packet drops as well as

a well-established ITU X.25 checksum used for packet corruption detection (MAVLink, 2019).

MAVLink consists of two versions, where MAVLink v1.0 contains a minimum of 8 bytes per packet sent (including start sign and packet drop detection) and MAVLink v2.0 containing 11 bytes, allowing for a more extensible protocol. MAVLink v1.0 can support up to 255 vehicles all running concurrently, where each vehicle will be assigned a vehicle ID (ranging from 1 to 255).

A packet transmitted via MAVLink v1.0 can be described by referring to Figure 2.3.

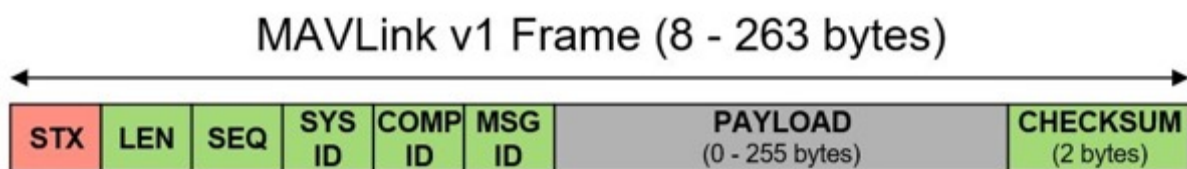


Figure 2.3: MAVLink v1.0 packet representation (Koubaa, et al., 2015)

The following table explains what each byte represents in the packet. This byte information was obtained from MAVLink (MAVLink, 2019).

Table 2.1: Description of each byte in MAVLink v1.0 protocol (MAVLink, 2019)

Byte Index	Content	Explanation
0	Packet start marker	Protocol-specific start-of-text (STX) marker used to indicate the beginning of a new packet.
1	Payload length	Indicates the length of the following payload section.
2	Packet sequence number	Used to detect any packet loss.
3	System ID	Represents the ID of the vehicle that is receiving the message.
4	Component ID	Represents the ID of the component that is sending the message.
5	Message ID	Represents the ID of the message type in the payload.
6 to (n+6)	Payload data	The message data. This depends on the message type (i.e. Message ID).
(n+7) to (n+8)	Checksum	Used to verify that the sender and receiver have perfectly understood a message.

MAVLink is built for hybrid networks to send high-rate data streams from data sources (commonly UAVs) to data sinks (commonly ground stations). This hybrid design pattern consists of a topic mode (publish-subscribe) and point-to-point mode. The topic mode is used to save bandwidth by sending a protocol, but will not emit a target system and component ID. A typical example of a command being used over this protocol would be to change a flight mode (e.g. position hold, acro, etc.). The point-to-point mode uses the target ID and target system when sending the message. Generally, when these fields are used, it guarantees delivery of the message. An example of when this protocol is used is when missions, parameters or commands are being sent to the aircraft (MAVLink, 2019).

2.7 Robot Operating System

When it comes to developing software for robots, it can often be quite challenging, particularly as the scale and scope of robotics continue to grow. Often, robots can have a wide variety of hardware, which often leads to a struggle of reusing code that was previously developed for other hardware. It can also be quite daunting if new code needs to be developed for these robots from scratch due to the code needing to contain a deep stack starting from driver-level software. (Quigley, et al., n.d.). To help with these challenges, Robot Operating System was developed.

Robot Operating System (ROS) Is a flexible framework that is used for writing and developing robot software. ROS consists of a collection of tools, libraries and conventions that aim at simplifying a task of creating complex and robust robot behaviour across a wide variety of robotic platforms (ROS, 2019a). One of the benefits of ROS is that the software is open-source. This allows for many people to use the framework, develop code and libraries and share it amongst the ROS community. An example of a package that is available to be used with ROS is MAVROS. This package is used to convert the ROS commands that are developed into commands that can be used on the MAVLink protocol, which will result in sending commands to a potential UAV.

Table 2.2 is a list of terms commonly used when using ROS. These terms will be commonly used throughout this research.

Table 2.2: Terms frequently used when using ROS

Term	Description of the term
Nodes	It is an executable that uses ROS to communicate with other ROS nodes (ROS, 2019b).
Messages	A ROS data type used when subscribing or publishing to a topic (ROS, 2019b).
Topics	Nodes can either publish messages to a topic or subscribe to a topic to receive messages (ROS, 2019b).
Publisher	A message that is published to a topic for other Nodes to access.
Subscriber	A topic that is subscribed to, where a message is able to be received from other nodes.
Service Server	Allows for two way transport of messages, where the server will receive a request, perform a task and send a reply.
Service Client	Allows for two way transport of messages, where the client requests a message and waits for a reply.
Workspace	It is the directory (folder) where all packages can be created or modified.
Package	It is the software (files) that is used or developed within ROS for a project.
Launch File	A file that will open all the nodes specified in the file as well as assign parameters certain values where required.

2.8 Euler Angles & Quaternions

Euler angles and quaternions are both used to represent a body's rotation or orientation. However, their methods of representing rotations are different. Euler angles are able to represent a 3D orientation of an object by using a combination of three rotations about different axes (CHRobotics, 2019a). These rotations all take place on a fixed coordinate frame.

Quaternions are also able to represent a 3D orientation of an object. However, they use a combination of a real number (scalar number) and complex numbers (imaginary numbers). The scalar part is represented by 'w' and the imaginary part is represented

by 'x', 'y' and 'z' (AnimMotion, 2019), resulting in a four dimensional vector [w, x, y, z] or [x, y, z, w].

Euler angles are easier to interpret than quaternions. However, the benefits of using quaternions are as follows:

1. Quaternions are not affected by gimbal lock whereas Euler angles can be affected. Gimbal lock is when a degree of freedom is lost due to two of the axes rotating around the same axis and aligning themselves (AnimMotion, 2019).
2. There is less computational processing required, due to quaternions only needing to be represented by a four element vector while Euler angles have a 3x3 matrix representation. Quaternions also require less memory space in comparison to Euler angles (AnimMotion, 2019).

Because of the gimbal lock, quaternions are preferred to be used in ROS to describe the orientation of an object.

2.9 Attitude of an Aircraft

The attitude (orientation) of an aircraft can be manipulated about the aircraft's current frame by making use of three types of rotation. These are:

1. roll (rotation about the x axis)
2. pitch (rotation about the y axis)
3. yaw (rotation about the z axis)

A representation of these rotations can be found in Figure 2.4.

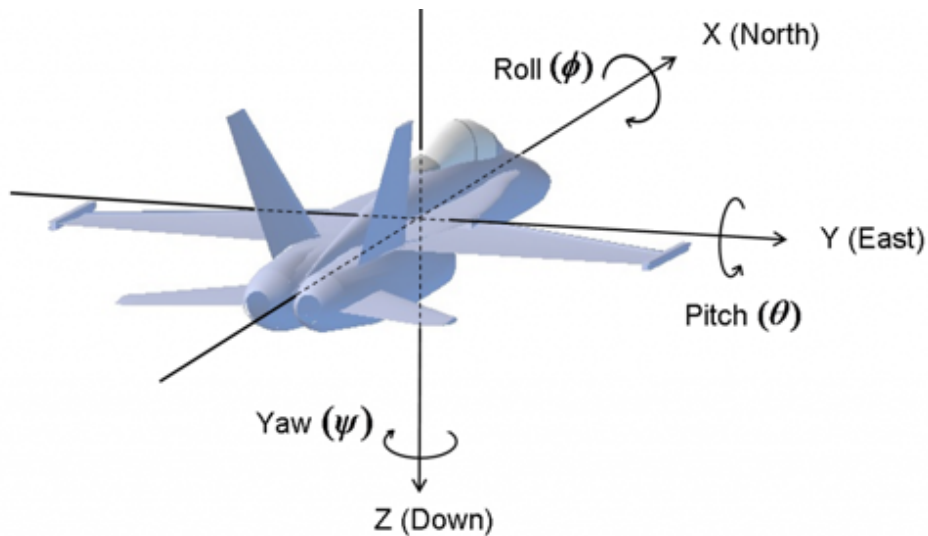


Figure 2.4: Rotations about the aircraft's current frame (CHRobotics, 2019b)

2.10 Vision

This section will be separated into two categories:

- Object detection, recognition & tracking
- Darknet & YOLO

2.10.1 Object detection, recognition and tracking

Object detection refers to detecting the presence of a particular object in a given frame, where the object in the frame is unknown (Howse, et al., 2016; Shipitko, 2017).

Object recognition is the process of identifying an object in a given frame (Howse, et al., 2016). For example, an object recognition system would be able to detect that a person and a dog can be found in a particular frame.

Object tracking is the extraction of the motion of an object from a sequence of images estimating its trajectory (IGI Global, 2019).

2.10.2 Darknet and YOLO

Darknet is an open-source neural network framework that was written in C and CUDA (Redmon, 2013). The framework supports both Central Processing Unit (CPU) and Graphics Processing Unit (GPU) computation, meaning that the processing of the neural network can be performed either on the computer's main processor or by using

the graphics card. You Only Look Once (YOLO) is a real-time objection detection system that was developed using the Darknet framework (Redmon, 2013). Most object detection systems use some sort of classifier in its detection process, whereas YOLO has more of a regression approach. The way YOLO works is by applying the neural network to a full image. The neural network will divide the image into regions and will predict bounding boxes and probabilities for each region. The bounding boxes are then weighted, based on the predicted probabilities that were determined (Redmon, 2013). The detection on the image is performed in one run of the algorithm, hence the name You Only Look Once. An example of the detection software being performed on an image can be referred to in Figure 2.5.

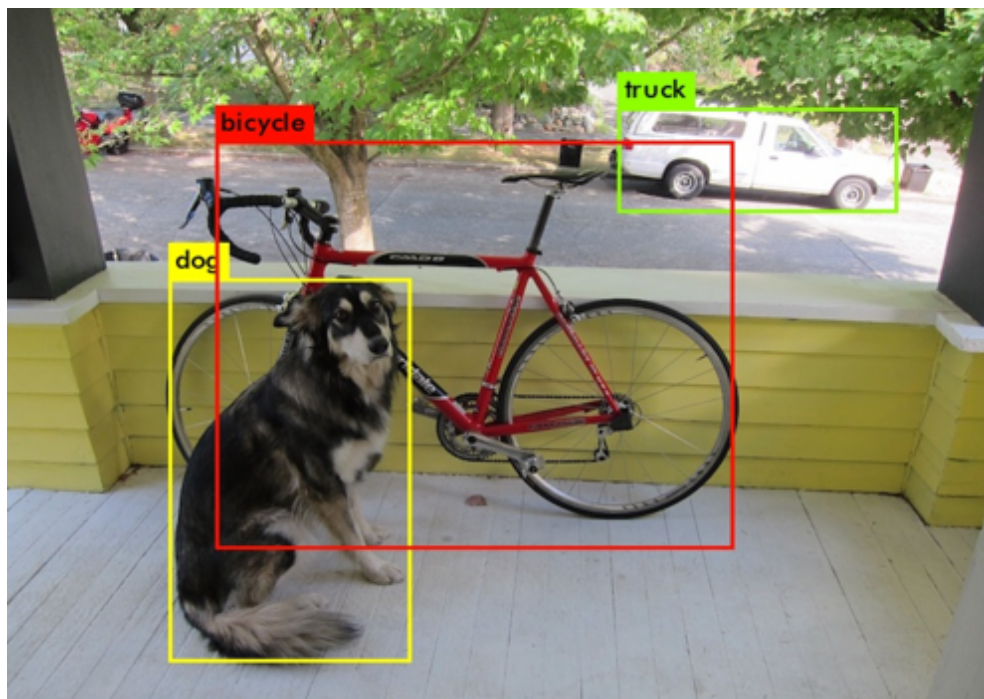


Figure 2.5: YOLO being used to detect multiple objects within a frame

The latest version available is YOLO version 3. This version has multiple models available to be used. These models vary, depending on what system the neural network is running on. For instance, there is model called Tiny YOLO available, which allows for the model to be run on low processing devices, such as a cell phone or a tiny computer (e.g. NVIDIA Jetson TX2). The current models available have been trained to identify 80 objects/classes, where it's based on the Common Objects in Context (COCO) dataset. However, the neural network can be trained to detect an

object of your choice. A list of 80 objects that are currently identifiable based on the COCO dataset can be found in Appendix 2.1.

2.11 Fiducial Markers

Many computer vision applications, robot navigation and augmented reality often require pose estimation (position and orientation estimation) within their applications. Pose estimation is based on finding correspondences between points in the real environment and their 2D image projection. To help with this, a fiducial marker can be used, which is a binary square marker that provides enough correspondence to obtain a camera pose. Their inner binary codification makes them specially robust, which helps with the possibility of applying error detection and correction techniques (OpenCV, 2019).

Some examples of fiducial markers can be seen in Figure 2.6.

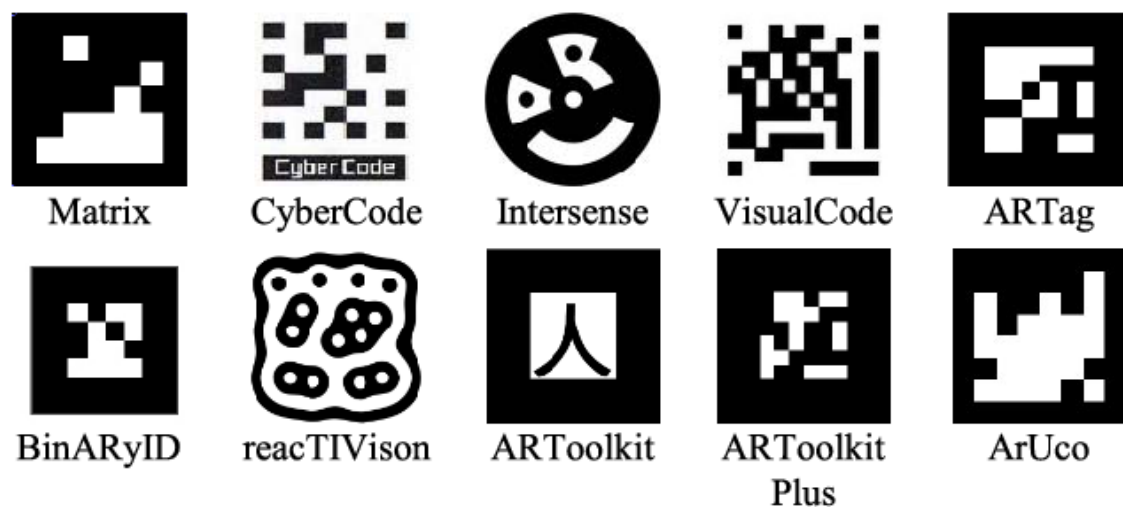


Figure 2.6: Examples of fiducial markers (Mostashiri & Dhupia, 2018)

One of most common types of fiducial markers used is the ArUco marker. It is a synthetic square marker that consists of a wide black border and an inner binary matrix which represents the marker's identifier (ID). The black border helps with faster detection within the image captured (OpenCV, 2019).

There are a number of dictionaries available for ArUco markers, all depending on the size of the internal matrix. This size of the matrix determines the number of bits for

the marker. For instance, if a marker has an internal matrix of 4x4, the marker will have a size of 16 bits. Within a specific dictionary, no two markers will look the same. For example, if a dictionary of 5x5 is used, the marker with an ID of 1 will look different to a marker that has an ID of 2.

2.12 Current Implemented Solutions

The following are current implemented solutions available or currently being developed solutions, where unmanned aerial vehicles are being used in surveillance systems.

2.12.1 The Sunflower System

The Sunflower System is a system that is currently being developed by Sunflower Labs. They claim their system will be able to sense and deter unwanted visitors before they reach your door. The system can be referred to in Figure 2.7. The system consists of three key components, which are the sunflowers, the bee and the hive (Sunflower Labs Inc., 2019).

- Sunflowers – These are sensors that are placed around the property which also help with lighting up the area. They are used to alert the user of unusual activity. The sensors are able to identify people, animals and cars.
- The Bee – This is an autonomous UAV that flies around the property. It has a camera to capture and live stream precisely everything that is currently happening at the time. The bee, which is normally guided by the sunflowers' sensors, can also be directed to specific locations on the property.
- The Hive – This is a self-charging, weatherproof home for the bee. Since the system has the sensors within the sunflowers, the bee isn't required to offer constant camera surveillance. When the bee isn't flying around, it will be docked safely within the hive, protecting the bee from the weather. The system's computer is also placed within the hive.



Figure 2.7: The Sunflower System (Sunflower Labs Inc., 2019)

2.12.2 Nightingale Security

Nightingale Security have developed their own surveillance system where they make use of a multirotor aircraft. Their fully autonomous system is able to fly patrols during the day and night as well as in the rain and snow (Nightingale Security, 2019). The aircraft is referred to as Blackbird, which has been named after the historic SR-71 Blackbird of the United States Air Force (Nightingale Security, 2019). This aircraft has a maximum flight time of 33 minutes and can be fully recharged within 45 minutes. The aircraft is able to perform scheduled autonomous patrols around a property, respond to any alarms that have been triggered as well as even perform a manual flight mission in the event of a crisis occurring. During flight time, a live video feed is transmitted to view what the aircraft is seeing. In one of the videos on their website, Nightingale Security demonstrates the aircraft's ability to detect human activity. When the aircraft is not being deployed, it lands and charges within a base station. The base station, which has a rugged, weatherproof design, contains networked computers that share critical flight information from the aircraft (Nightingale Security, 2019). The system can be found in Figure 2.8, where the aircraft is currently positioned on the docking station.

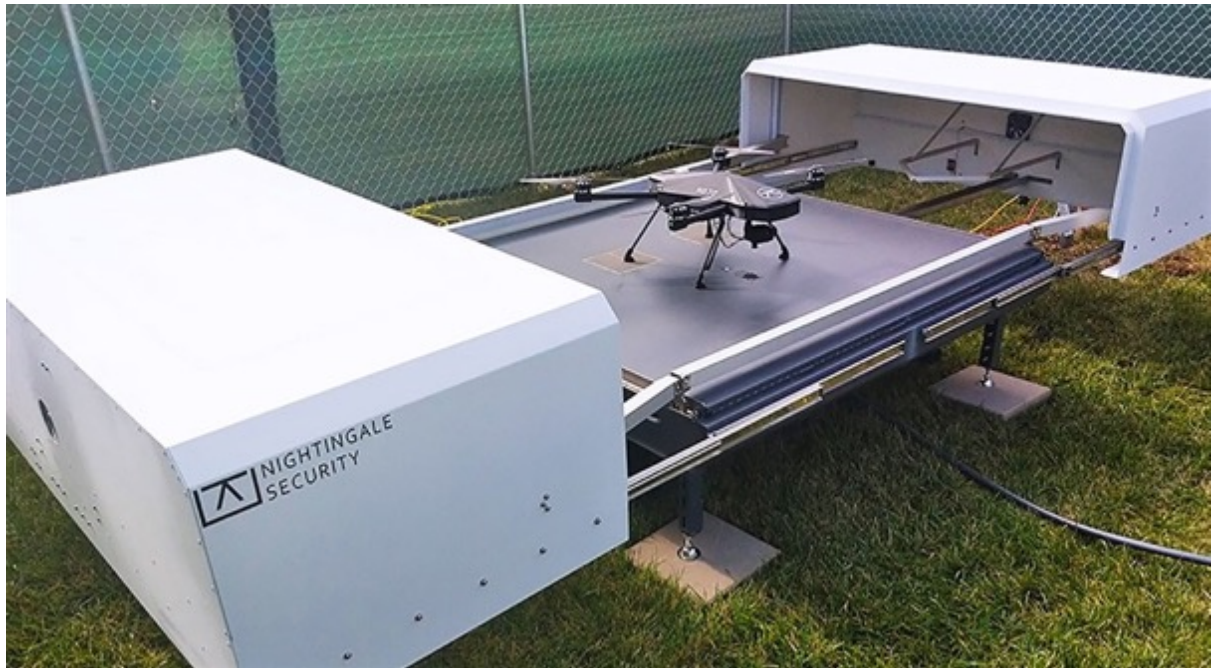


Figure 2.8: Nightingale Security system (ISC West, 2019)

Nightingale Security's system has been approved by the Federal Aviation Administration (FAA) in the United States of America for performing night time and multi-drone operations (Nightingale Security, 2019). They offer a monthly or annual subscription model, where customers don't need to purchase or maintain any equipment.

2.13 Conclusion

This chapter shed light on a number of topics that will be used in this research. Due to the nature of the research topic, the equipment and software that was researched was fairly new technology, resulting in minimal information being available in journal papers. The information included in this chapter was therefore primarily obtained from websites. The typical components that can be found on a multirotor aircraft were discussed. Hardware, such as the Arduino and the Pixhawk, were introduced and their purposes explained. Software, such as the autopilot software and the ground control station software, were explained. Other topics, such as Darknet's YOLO, fiducial markers, Euler angles, quaternions and the attitude of an aircraft were also explained. The significance of using ROS was discussed. Current solutions were mentioned, showing how a complete system should be implemented. Finally, the crime statistics for South Africa were revealed, stressing the importance of developing

a system to help reduce the overall crime in the country. The next chapter is the System Architecture, which will discuss the entire design of the system.

Chapter 3: System Architecture

This chapter will be divided into three sections. The first will give a basic overview of the dynamics of the aircraft; the hardware architecture of the research will be discussed in the second; and the software architecture in the third. The second section will include comments on the numerous changes that were implemented. The configuration of the hardware will also be discussed. The third section will make mention of all the software that was used throughout the system.

One of the keys aspects for this research was to keep costs to the minimum. Since the research was still in the developmental stage at the time of writing, there were likely to be a number of crashes occurring during the testing phase. If a crash were to occur and some of the components were to break, the replacement components would need to be inexpensive.

3.1 Multirotor Aircraft Dynamics

In order for a multirotor aircraft to fly, it needs to overcome and oversee three crucial factors: drag, thrust and lift (Kadamatt, 2017). Other factors which have an effect on the way in which it flies are hovering, increasing and decreasing in altitude and movement along its yaw, pitch & roll axes.

3.1.1 Drag

The word 'drag' is essentially a mechanical force that opposes the motion of any object through a fluid (Kadamatt, 2017). Since the motors of a multirotor aircraft pass through air, it is called 'aerodynamic drag' (as opposed to the word, 'hydrodynamic drag' used to represent objects passing through water) (Kadamatt, 2017).

This aerodynamic drag on the rotors is generated due to the difference in velocity between the rotors and the air around them (Kadamatt, 2017). The drag force is only applied to the multirotor aircraft when it is in motion (either in a vertical, horizontal or rotational direction). The drag force can be seen in Figure 3.1 & Figure 3.3.

In order for a multirotor aircraft to rise off the ground and fly, it will have to overcome the drag force as well as the overall weight of the aircraft (Kadamatt, 2017). This may be overcome, depending on the thrust generated by the motors.

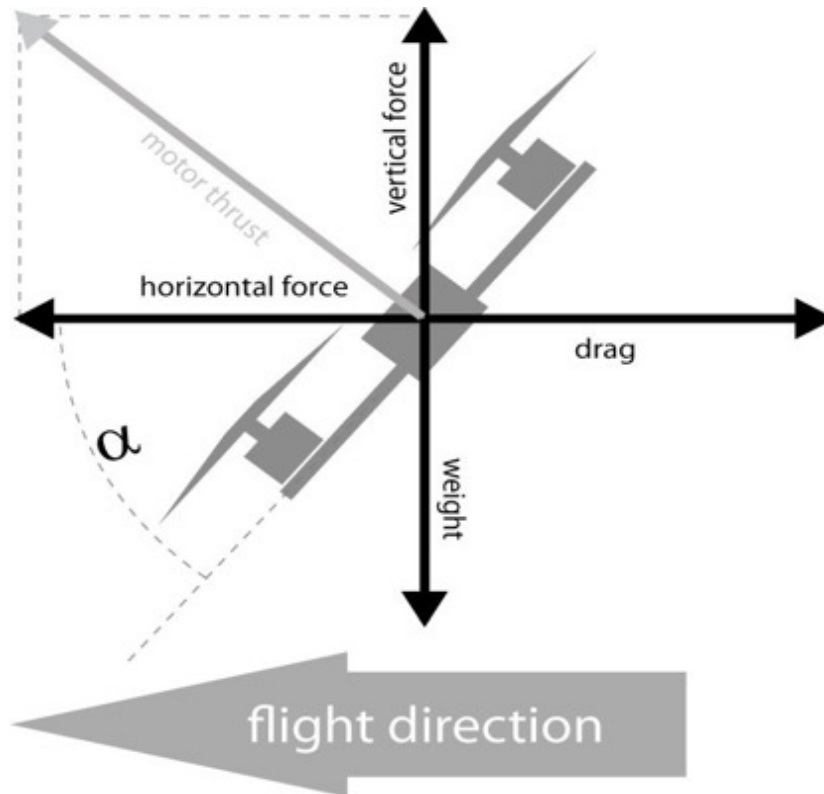


Figure 3.1: Forces acting on a multirotor aircraft

3.1.2 Thrust

Thrust stands for the force generated by the propellers that are attached to the motors. The thrust force is used to overcome the drag force generated, as well as the overall weight of the multirotor aircraft. The thrust force generated is not the main force responsible for the rising of the multirotor aircraft. The lift force is responsible for this. (How Things Fly, 2005). The thrust is the force which allows the motors to travel through the air, overcoming the resistance caused by the drag. The thrust force may be seen in Figure 3.1.

3.1.3 Lift

The lift of the multirotor aircraft is the force that works against the weight of the aircraft, when being lifted into the air. The following are responsible for the lift on a wing (on the propeller):

1. Newton's Third Law of Motion – For every action, there must be an equal and opposite reaction. This force will generate lift at the bottom of the wing because the mass of the air is pushed downwards and backwards (Kadamatt, 2017).
2. Bernoulli's – The pressure difference between the air at the top and the bottom of a wing, due to the Coanda Effect, will generate a lift towards the lower pressure, which is present at the top (Kadamatt, 2017). This theorem is still being tested to date. The effect can be seen in Figure 3.2.

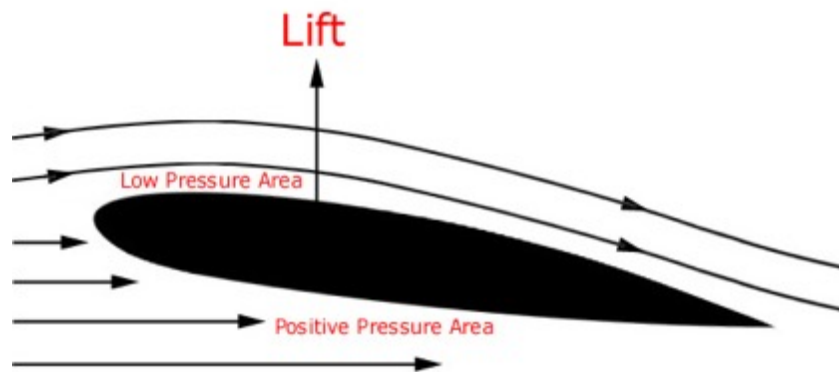


Figure 3.2: Coanda Effect (Aviation History, 2015)

The propellers attached to the rotors will generate the lift force, using similar principles as those mentioned above (which is pushing air downwards as well as the difference in the air pressure). In order for the aircraft to rise, be able to hover and more importantly, fly, the lift force must be greater than the weight of the aircraft. The lift force can be seen in Figure 3.3.

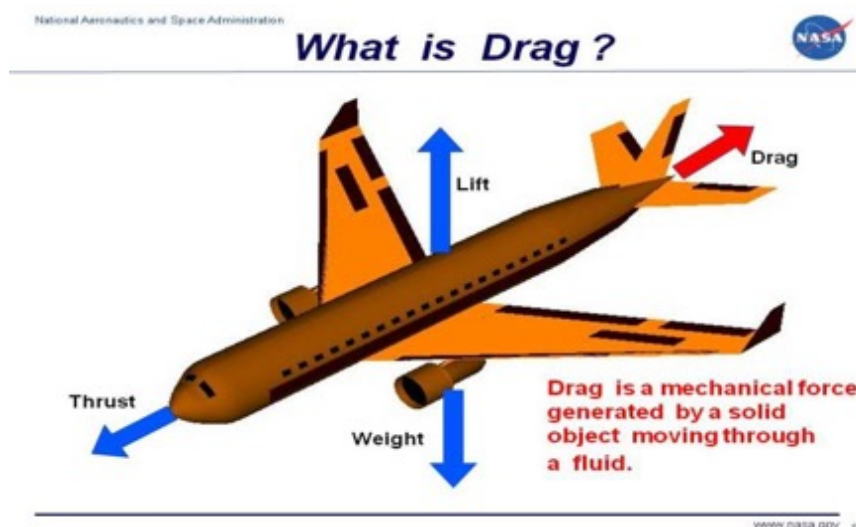


Figure 3.3: The forces applied to an aircraft (NASA, 2015)

3.1.4 Flying

The way in which a multirotor aircraft works is based on how the brushless motors work. The rotational part of the motor is referred to as the rotor. As specified earlier, a multirotor aircraft is a helicopter that consists of numerous motors that work together to make the helicopter fly in certain directions. The following description will be based on a hexacopter aircraft, which is a multirotor aircraft that contains six arms and six motors. The way in which these six motors work depends on the direction in which the motors are spinning. Each of the six motors will be attached to each of the six ends of the arms that are attached to the frame. Three of the motors will spin in a clockwise direction (CW), while the other three will spin in a counter clockwise direction (CCW) (Allen, 2014). By doing so, any tendency of the motors to cause the aircraft to spin in the air uncontrollably, due to the torque generated by the motors, will be cancelled out. All clockwise motors are positioned in a triangle shape, while the counter clockwise motors are also positioned in a triangle shape. Figure 3.4 below demonstrates motors 1, 3 and 6 spinning in a clockwise direction while motors 2, 4 and 5 are spinning in a counter clockwise direction.

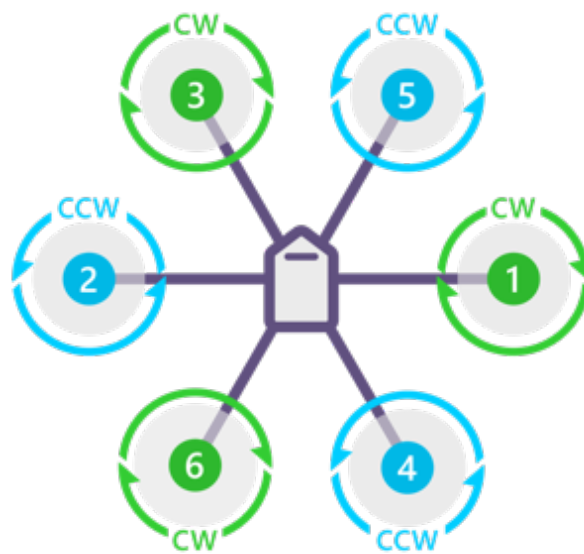


Figure 3.4: Direction of rotation for the brushless motors (ArduPilot, 2019)

At each of the motors, a torque is generated, which is in the same direction that its specific rotor is turning. The direction in which each of these rotors rotates will counteract this generated torque. The net torque will therefore equal zero (Kadamatt, 2017). This is the basis on how a multirotor aircraft is able to fly.

3.1.5 Hovering

For a multirotor aircraft to hover in mid-air at a fixed position, the following needs to take place (Kadamatt, 2017):

1. All the motors need to rotate at the same speed (calculated in revolutions per minute - RPM).
2. The rotational speed of the motors needs to be sufficient, allowing the multirotor aircraft to generate enough lift to counteract its own weight.
3. The torques created by each of the motors, which act against the frame of the multirotor aircraft, need to be cancelled out or else the aircraft will want to yaw in a specific direction.

3.1.6 Increasing and Decreasing Altitude

In order for a multirotor aircraft to gain altitude, all of the rotors are required to increase their rotational speed simultaneously. Conversely, for a decrease in altitude, the motors should decrease their rotational speed simultaneously. This increasing and decreasing in altitude is achieved by using the throttle & elevator control sticks on the transmitter (refer to the transmitter in section 3.2.6) (Kadamatt, 2017).

3.2 Hardware Architecture

This section will describe all the components that were used and installed throughout the research as well as the final hardware architecture.

3.2.1 DJI vs Self-developed Aircraft

There were two possible approaches in deciding what type of aircraft would be used. The first approach was to use a DJI Phantom 4 Advanced aircraft, where onboard software could be developed for the aircraft via DJI's Software Development Kit (SDK). The second approach was to develop an aircraft from scratch. There are advantages and disadvantages to both approaches. The advantages and disadvantages for selecting the DJI Phantom 4 Advanced can be found in Table 3.1 and the advantages and disadvantages for developing an aircraft can be found in Table 3.2.

Table 3.1: Advantages and disadvantages of using a DJI Phantom 4 Advanced

Advantages	Disadvantages
Top of the range off-the-shelf aircraft.	Expensive to purchase.
Already has a functional vision system.	Replacement parts are expensive.
Attachments can be added to the aircraft (e.g. FLIR thermal camera).	Not much physical space available to add additional hardware.
Has a flight mode available to follow a moving object (where the object needs to be selected via DJI's mobile app).	Only the SDK is available. Cannot run any choice of software on the aircraft.
DJI's SDK is available.	

Table 3.2: Advantages and disadvantages of developing an aircraft

Advantages	Disadvantages
Completely customizable.	Not a ready off-the-shelf aircraft that can operate straight away.
A Pixhawk flight controller can be used.	Many test runs will be required to get the aircraft running efficiently.
Multiple components can be added to the aircraft (e.g. companion computer, camera, ultrasonic sensor, LIDAR, etc.).	
Components are inexpensive compared to DJI's components.	
Can lead to future development projects (e.g. developing a docking / charging station for the aircraft).	

It was decided that developing an aircraft would be the preferred choice, mostly due to the aircraft being completely customizable. At the time of deciding, a second-hand hexacopter was offered and purchased from the advisor, Mr Paul Mooney. This decision was made due the aircraft already having numerous components pre-installed from a previous project he worked on. It was also easily available and allowed for immediate hands on the aircraft. The aircraft was a hexacopter, as opposed to a traditional quadcopter. This means that if a motor or ESC were to fail, the aircraft

would still have five motors running, which would allow for the aircraft to still fly partially and be allowed to land safely.

The purchased second-hand aircraft came with the following components:

1. Frame & extra legs
2. Six brushless motors
3. Six propellers
4. Six electronic speed controllers

Before discussing the components, it must be noted that in order to use ROS with the aircraft, a number of components were required to be added to the aircraft. They were:

1. Companion computer
2. USB Hub
3. 5V Regulator
4. Camera
5. Wi-Fi Module
6. FTDI Cable

More information on these components will be discussed later in the chapter.

3.2.2 Frame

The frame of the second-hand hexacopter aircraft was a generic DJI F550 Flame Wheel. An example of this frame can be found in Figure 3.5. This frame contained six arms, which means that the aircraft would make use of six motors and six electronic speed controllers (ESC) to provide the thrust required by the aircraft. The frame also consisted of a power distribution board (PDB), which allowed for the electronic speed controllers to be connected to the battery. Extra attachments that were included in the purchase of this frame were four legs, which provided the aircraft with extra height off the ground as well as allowed the aircraft to have extra attachments beneath its fuselage.



Figure 3.5: DJI F550 Flame Wheel (Build Your Own Drone, 2019)

3.2.3 Flight Controller

There were numerous changes made to the flight controller, the reasons for which will be discussed below. It must be noted that only Pixhawk flight controllers were used throughout this research because:

- A Pixhawk flight controller is an autopilot flight controller, which meant that it could perform autonomous flights.
- The flight controller is open source, which meant that any open source firmware (e.g. ArduPilot, PX4, etc.) could be uploaded onto the board.
- Components could easily be plugged into the flight controller, allowing for disconnecting and attaching of new components to the board with ease.
- The flight controller is one of the most popular flight controllers in use, meaning that there was a lot of information available to developers.

3.2.3.1 Generic Pixhawk 1

A second-hand generic Pixhawk 1 was first used and installed. This flight controller was used due to its being available locally, allowing for immediate testing. However, it was replaced after two weeks into the research. A general idea of what the generic Pixhawk resembles can be seen in Figure 2.2.

this Pixhawk was that it was capable of using real-time kinematic (RTK) positioning to help with accurate positioning. However, this equipment was rather expensive, so the standard Here+ GPS module (refer to Figure 3.7(b)), that came with purchasing the flight controller, was installed alongside a power module to power the Pixhawk.



Figure 3.7: (a) Pixhawk 2 Cube (GetFPV, 2019) (b) Here+ GPS module (Heli Engadin, 2019)

3.2.3.4 Original Pixhawk 1

During testing of the Pixhawk Cube, a problem occurred which prevented the flight controller from arming. Arming refers to the aircraft passing all its safety checks and allowing the brushless motors to spin at an idle speed, where the aircraft won't take-off. All aircraft need to be armed first before commencing their flights. The problem was that the magnetometer kept giving inconsistent values to the Pixhawk. After conducting some research on the internet, it was discovered that the reason for this was that the Here+ GPS module contained two GPS boards within the module, where the one was placed 180° from the other GPS board. This meant that the one GPS board was facing towards the tail of the aircraft. The discrepancy caused an issue with the PX4 firmware, resulting in the magnetometer being inconsistent. It must be noted that this problem does not occur with the ArduPilot firmware.

As a result, the flight controller needed to be replaced with an original Pixhawk 1 flight controller, which was manufactured by 3DR. This flight controller was chosen due to

its being locally available as it belonged to a colleague. It resulted in a quick and efficient solution. The Pixhawk can be seen in Figure 2.2.

3.2.4 Electronic Speed Controller

The Electronic Speed Controllers (ESCs) that were included with the second-hand aircraft were Afro-ESC 30A with SimonK firmware installed on them. The ESC can be referred to in Figure 3.8. SimonK firmware, (which was developed by Simon Kirby), was beneficial for use on multirotor aircrafts due to its helping the ESCs offer a faster response than your typical ESCs, resulting in the brushless motors reacting quicker to changes in the throttle.



Figure 3.8: Afro-ESC 30A ESC (RC Groups, 2013)

3.2.5 Brushless Motor & Propellers

The motors that came with the second-hand aircraft were Prop Drive 28-26s 1000Kv brushless motors, which can be referred to in Figure 3.9(a). The Kv refers to the constant velocity of the motor, which is measured by the number of revolutions per minute that a motor turns when one volt is applied with no load attached to the motor (Reid, 2016). That means that if a three cell 11.1V LiPo was attached to the motor with no load, the motor would spin at 11 100 rpm (1000×11.1).

The brand of propellers received were APC and the size of the propellers were 10 x 4.7 inches (refer to Figure 3.9(b)). The size means the diameter of the propeller was 10 inches and the pitch of the blade was 4.7 inches.



Figure 3.9: (a) Prop Drive brushless motor (HobbyKing, 2019) (b) APC 10 x 4.7 inches propeller (Hobby-Miracle, 2019)

3.2.6 Transmitter & Receiver

In order to fly and control the aircraft manually, a transmitter and receiver needed to be used. The transmitter is a remote controller that the pilot holds and uses to direct the aircraft. The receiver would need to be installed onto the aircraft to receive the commands from the pilot. The type of transmitter that was used in this case was a FrSky 2.4Ghz ACCST Taranis Q X7. The receiver that was attached to the aircraft's fuselage was a FrSky 2.4Ghz ACCST X8R. The transmitter and receiver can be found in Figure 3.10.



Figure 3.10: (a) FrSky Taranis Q X7 transmitter (GetFPV, 2019) (b) FrSky X8R receiver (PorcupineRC, 2019)

3.2.7 Battery

To power the motors, ESCs, Pixhawk, receiver and other components on the aircraft, a Lithium Polymer (LiPo) battery was required. Two identical Gens ace 3300mAh 11.1V 25C 3S1P batteries were purchased, where they were connected in parallel, to provide an overall capacity of 6600mAh. The battery can be seen in Figure 3.11.



Figure 3.11: Gens ace 3300mAh LiPo battery (Unmanned Tech, 2019)

3.2.8 Flight Controller's Extra Components

Included in the Pixhawk 2.4.6 kit that was purchased (as mentioned in section 3.2.3.2) was a GPS module, GPS stand and a 433Mhz telemetry, where all three components were installed onto the aircraft. Inside the GPS module was a NEO-M8N GPS and a magnetometer. The GPS module, GPS stand and the 433Mhz telemetry can be referred to in Figure 3.12.



Figure 3.12: (a) GPS Module and GPS Stand (Amazon, 2019) (b) 433Mhz telemetry (Readytosky, 2019)

However, when the flight controller was switched to the genuine Pixhawk 1, the aircraft's current GPS module and 433MHz telemetry were swapped out for the GPS module and 915MHz telemetry that came along with the genuine Pixhawk 1.

3.2.9 Gimbal

To install a camera onto the aircraft, a gimbal was required to help stabilize the video footage. This is done by allowing the camera to keep facing its desired direction and allowing the aircraft to move and twist around. The gimbal that was purchased was a Tarot ZYX T-3D V 3-axis gimbal, which was designed to carry a GoPro Hero 5 camera. The reason for purchasing this gimbal, which can be referred to in Figure 3.13, was due to there being a GoPro camera available in the laboratory at the time of testing the aircraft. However, the end goal was actually to use the gimbal to carry another type of small camera that would be connected to a potential companion computer (refer to section 3.2.11).



Figure 3.13: Tarot ZYX T-3D V gimbal (Tarot, 2017)

While experimenting with some of the settings on the software for the gimbal, the gimbal controller module overheated and caused the module to become faulty. The expected cause of the overheating was that the USB port of the laptop that the gimbal was plugged into was a unique USB port that allowed any device to charge from the port while the laptop was powered off. Unfortunately, a replacement module was not available to be purchased on its own, so a new gimbal had to be purchased. This time round, a different version gimbal was purchased, which was a Tarot T4-3D. This gimbal was designed for a GoPro Hero 3, Hero 3+ and Hero 4 camera, whereas the

former gimbal was only designed for a GoPro Hero 5. This gimbal can be seen in Figure 3.14.



Figure 3.14: Tarot T4-3D Gimbal

3.2.10 Companion Computer

A companion computer was used on the aircraft to send and receive information to and from the flight controller. It was decided that an Odroid XU4 companion computer, which can be referred to in Figure 3.15(a), would be chosen because:

- It was small enough to fit onto the aircraft.
- The purchasing cost for a computer, computer case, power brick and eMMC module was below R2 000.
- It had a low power consumption.
- It allowed for Linux Ubuntu to be installed onto the computer.
- It allowed for peripherals (e.g. webcam, Wi-Fi module, etc.) to be plugged into the computer.

To store the operating system and the files for the computer, the Odroid had two options available. The choice was between using either a Micro SD card or an eMMC module. It was decided that a 32gb eMMC module would be used as it had faster read and write speeds compared to the micro SD card. A chart demonstrating the difference in speeds between a Micro SD class 10 card, a Micro SD UHS-1 card and an eMMC 5.0 module can be referred to in Figure 3.15(b). As mentioned, a case for the Odroid was also purchased.

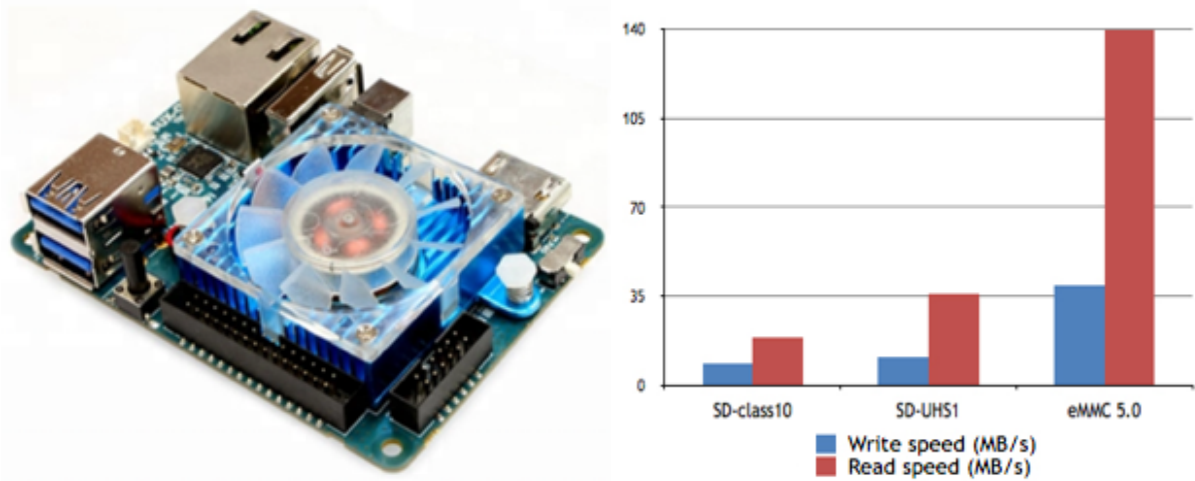


Figure 3.15: (a) Odroid XU4 (Hardkernel, 2019a) (b) Comparing the read and write speeds (Hardkernel, 2019b)

3.2.11 Camera

A camera was required to be used for the vision system. Since this research was still in its developmental stage, it was decided that a low cost Logitech C920 HD webcam would be used instead of a more industry standard camera. The camera, which can be referred to in Figure 3.16, had the capability of filming footage in 1080p (full HD) at 30 frames per second (FPS). Multispectral cameras and FLIR thermal cameras were considered for use for this research. However, it was regarded as an unnecessary expense. A big advantage of using a webcam was that it allowed for the digital video feed to be used by the companion computer via one of the computer's USB ports.



Figure 3.16: Logitech C920 webcam (Logitech, 2019)

3.2.12 Extra Components

A USB hub was required to be installed onto the aircraft. This was due to there being a shortage of USB ports available on the Odroid, when a keyboard and mouse were also plugged into the Odroid. In addition, when plugging in the USB ports, it was found that the Odroid was not providing enough power to the peripherals. Therefore, the USB hub would need to have an external power source to help provide enough power to the peripherals. The USB hub that was eventually installed onto the aircraft was a Unitek USB3.0 4-Port Hub. This hub can be referred to in Figure 3.17.



Figure 3.17: Unitek USB3.0 4-Port Hub (Unitek, 2019)

In order to connect the Odroid and the USB hub to the LiPo batteries, the LiPo's maximum peak 12.6V (4.2V per cell) needed to be stepped down to the necessary 5V required by the devices. A Hobbywing Universal Battery Elimination Circuit (UBEC) 5V at 8A (with a maximum of 15A) was purchased and connected to the LiPo batteries. The UBEC, which can be seen in Figure 3.18, is in essence a voltage regulator.



Figure 3.18: Hobbywing UBEC 8A (Hobbywing, 2015)

A Wi-Fi module and an FTDI cable were also purchased. The Wi-Fi module was connected to one of the USB ports on the USB hub that was connected to the companion computer. This Wi-Fi module was used to connect the Odroid computer to a network, which was established to connect the Odroid computer to a ground station computer. The Wi-Fi module was a standard Wi-Fi N module. The Future Technology Devices International (FTDI) cable is a USB to Serial converter, which was used to connect the companion computer to the flight controller. The serial side of the FTDI cable was cut off and a new end was soldered onto the wires so that the cable could be compatible with the Pixhawk flight controller. The FTDI cable and Wi-Fi module can be found in Figure 3.19.



Figure 3.19: FTDI cable (left) and Wi-Fi module (right)

An additional component that was used, but not placed onto the aircraft, was a D-Link DSL-2750U modem router. This router had the capabilities of using a 3G/4G modem to connect the network to the internet. (However, this feature was not used). For this research, the router was used to establish a network so that the companion computer could communicate with the ground station computer. Since the router would be placed outside during testing, it needed to be modified so that it could be powered from an external power supply and not from a typical wall outlet. The modification consisted of soldering two wires onto the router's board, while the other end of the wires were soldered to an XT60 connector so that a 3 cell LiPo battery could power the device. This modification can be seen in Figure 3.20.



Figure 3.20: Modified D-Link DSL-2750U modem router

3.2.13 Additional Modifications

A carbon rod was attached to the aircraft with two polystyrene balls placed on either end. This was attached to help the pilot determine the current direction of the aircraft during flight. Each ball was coloured differently to distinguish between the nose (front) and the tail (back) of the aircraft. The red and blue polystyrene ball with tinfoil wrapped around it was attached to the nose of the aircraft and the green polystyrene ball was attached to the tail. This setup can be referred to in Figure 3.21.



Figure 3.21: Polystyrene balls used to help with the direction of the aircraft.

However, after flying the aircraft, It was decided that the front polystyrene ball at the nose of the aircraft was no longer necessary and so the rod was cut in half. Only the half at the tail end was kept. An extra piece of carbon rod was placed between the two arms at the tail to secure the rod to the aircraft. This modification can be found in Figure 3.22.



Figure 3.22: Modification of keeping the tail polystyrene ball and supporting the rod

3.2.14 Designed Components

The following parts were designed in Autodesk Inventor, where they were made via a 3D printer and tested on the aircraft.

3.2.14.1 Gimbal Attachment v1

Two identical parts needed to be designed in order to attach the Tarot ZYX T-3D V gimbal (the first gimbal used) to the aircraft. Each part would need to allow for the gimbal to be hung from it by means of a rod, cut to size. The part itself would also need to be able to hang from the bottom plate of the aircraft's fuselage.

The bottom plate of the fuselage contained numerous holes, which were used to push the parts through and suspend each one with its own bolt. The designed part can be viewed in Figure 3.23.

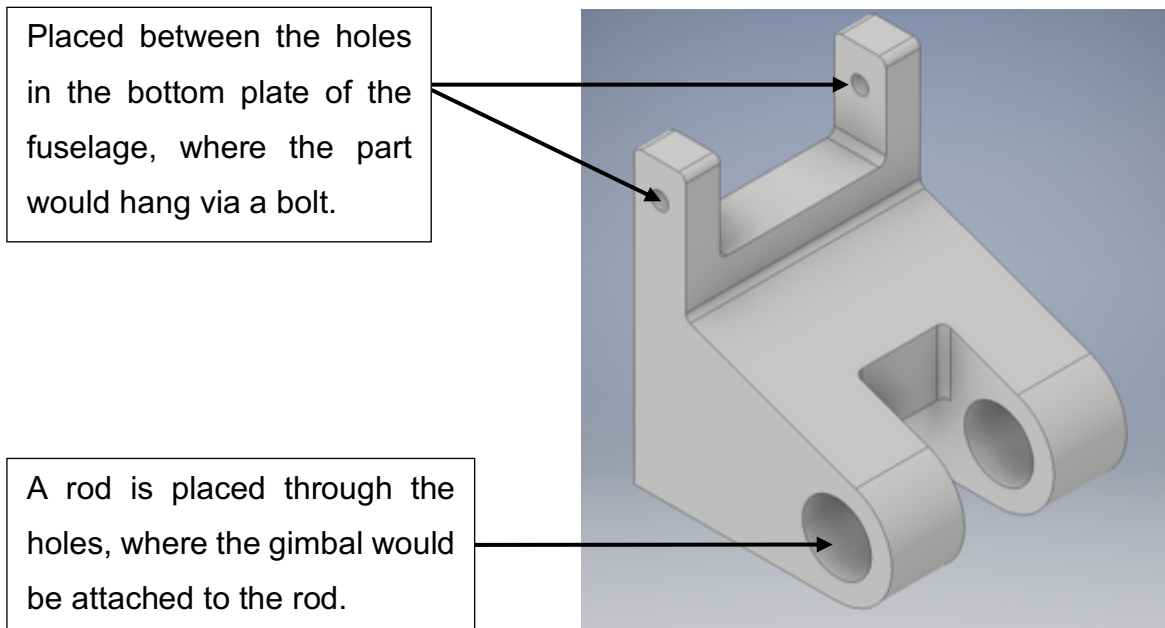


Figure 3.23: Part designed to attach the gimbal to the bottom plate of the aircraft's fuselage

The designed parts were fully functional. However, there were two concerns. The first was that there was too big a space between the bottom plate of the aircraft and the gimbal. The second was that, since the part was hanging from the aircraft, it allowed for extra movement of the gimbal, which could affect the stability of the gimbal. This meant that the part would need to be redesigned at a later stage. However, for the time being, the parts were sufficient and fulfilled their purpose. The CAD drawing of the part can be found in Appendix 3.1.

3.2.14.2 Gimbal Attachment v2

When the Tarot ZYX T-3D V was replaced with the Tarot T4-3D, a new part needed to be designed to attach the new gimbal to the bottom plate of the fuselage. This was due to the new gimbal having slightly different dimensions to the previous gimbal. With the new part came a new design, where instead of letting the designed part hang from the fuselage, the part would rather be fastened to the bottom plate, only allowing the gimbal to be hung. This would prevent the previous issues from occurring. The new designed gimbal attachment can be found in Figure 3.24. Once again, two identical parts were required to be printed. The CAD drawing of the part can be found in Appendix 3.2.

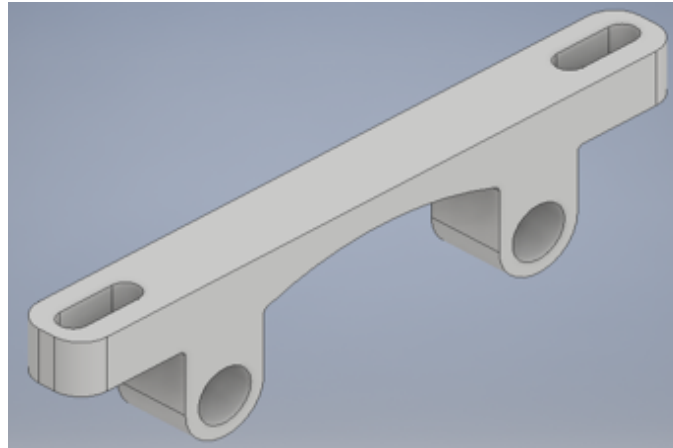


Figure 3.24: New part designed to attach the gimbal to the aircraft's bottom plate.

3.2.14.3 Leg Extension

When the first gimbal and the two parts were attached to the aircraft, another issue occurred. The four legs that were currently attached to the aircraft were not long enough to prevent the gimbal from touching the ground. To fix this, another part was developed that allowed the legs to be extended. This part, which was also 3D printed, can be referred to in Figure 3.25. The part was designed to be hollow at the top, which allowed for a portion of the bottom of the original leg to be placed within the part. The part was also designed to follow the curve of the leg. Since there were four legs on the aircraft, this meant that four parts were required to be printed. These legs were still used with the second gimbal. The CAD drawing of the leg extension can be found in Appendix 3.3.

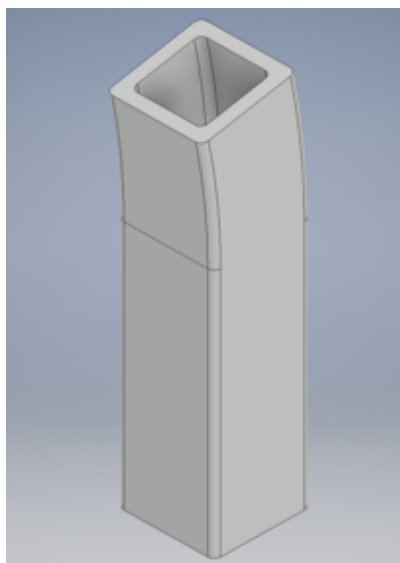


Figure 3.25: Part designed to extend the legs on the aircraft.

3.2.14.4 GPS Case

During one of the test flights (which consisted of the Pixhawk 2.4.6 flight controller, the first gimbal and the extension legs), the aircraft tipped over and crashed, resulting in the case of the GPS module as well as the wiring of the GPS module being damaged. To resolve this issue, a new GPS case needed to be designed. The designed case had a similar design to the original GPS case. The case consisted of two parts, where the top part had a concave shape and the bottom part had a cylindrical shape. Both parts had a cube shape hollowed out to allow for the GPS module to have a snug fit within the case. Since the GPS module needed to be positioned in a similar orientation to the flight controller, the outside of the concave part had an arrow designed on it to show the direction in which the GPS case needed to be positioned. The inside of the concave part had tiny legs designed along the border of the hollowed shape to allow the GPS module to be placed on the legs. A cylindrical hole was also designed on the side of both parts to allow room for the cable to pass through. These parts, which can be seen in Figure 3.26, were also 3D printed. The CAD drawing for the top and the base of the GPS case can be referred to in Appendix 3.4.

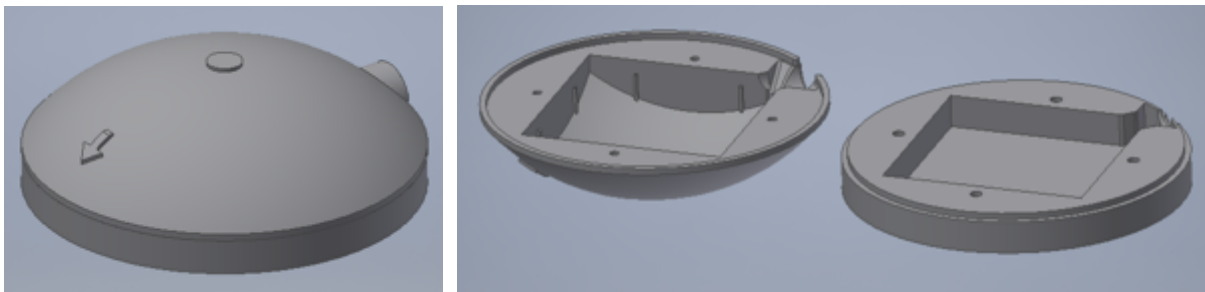


Figure 3.26: (a) GPS case designed in CAD (b) GPS case opened

3.2.14.5 Webcam Case

A part was designed to attach the webcam to the Tarot T4-3D gimbal. This part contained two cubes. The first cube was used to fill the space in the gimbal where the GoPro normally would have been mounted. The second cube was hollowed out to house the webcam, where it also contained a lid to secure the webcam inside the part. The part had a cylindrical hole cut out to allow the lens of the webcam to be outside of the part. To save time, Mr Martin Corlett, a colleague who worked in the same laboratory, was approached to design the part. It appears in Figure 3.27. The CAD drawing for the parts can be seen in Appendix 3.5.

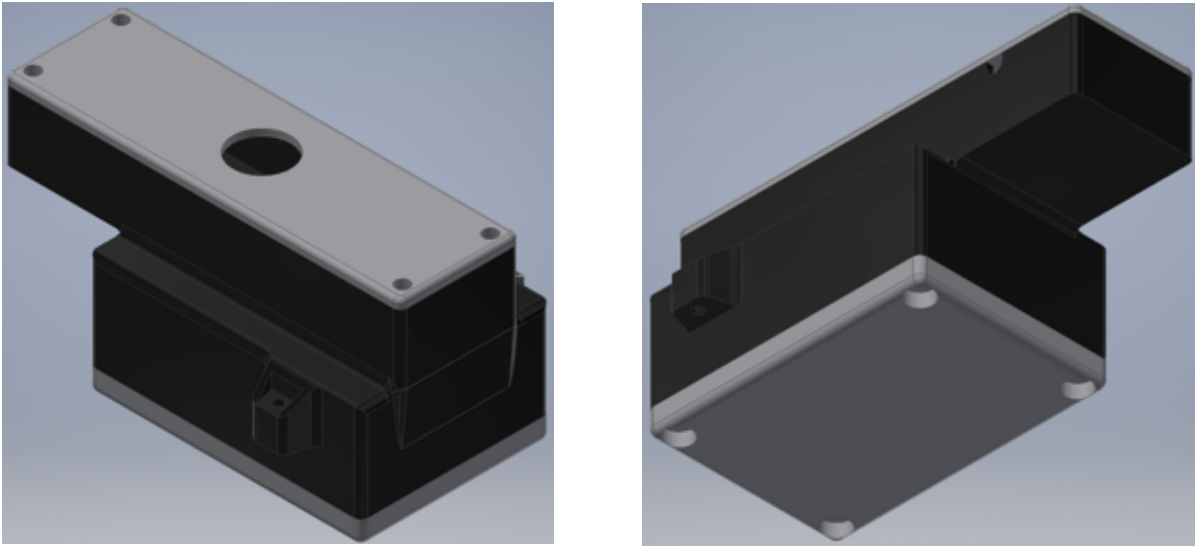


Figure 3.27: (a) the front of the webcam case (b) the back of the webcam case, which is fastened to the Tarot T4-3D gimbal

3.2.15 Final Hardware Components

Table 3.3 includes a list of components, as well as the quantity that were used, for the final design of the aircraft.

Table 3.3: List of components that can be found on the final design of the aircraft

Component	Name of component	Quantity
Frame	DJI F550 Flame Wheel	1
Flight Controller	Original Pixhawk 1	1
ESC	Afro-ESC 30A with SimonK firmware	6
Brushless Motor	Prop Drive 28-26s 1000Kv	6
Power Module	3DR power module	1
Receiver	FrSky X8R	1
Telemetry	3DR 915MHz telemetry	1
Gimbal	Tarot T4-3D	1
Companion Computer	Odroid XU4	1
USB Hub	Unitek USB3.0 4-Port Hub	1
Camera	Logitech C920 HD webcam	1
Voltage Regulator	Hobbywing UBEC 8A	1
Wi-Fi Module	Wi-Fi N Module	1
FTDI Cable	-	1

Table 3.4 shows all other necessary components that were used, but were not placed onto the aircraft.

Table 3.4: List of other components not directly used on the aircraft

Component	Name of component	Quantity
Transmitter	FrSky Taranis Q X7	1
Router	D-Link DSL-2750U modem	1
Ground Station Computer	Acer Predator 17 inch laptop & MacBook Pro 15 inch laptop	1

3.2.16 Final Aircraft Hardware Architecture

A block diagram for the final hardware architecture of the aircraft can be found in Figure 3.28. Please take note of the direction of the arrows as some of the components have a bidirectional (as in two-way) data transfer between them. The black arrows represent the data transfer between the components and the red arrows represent the power being provided straight to the components from the LiPo batteries.

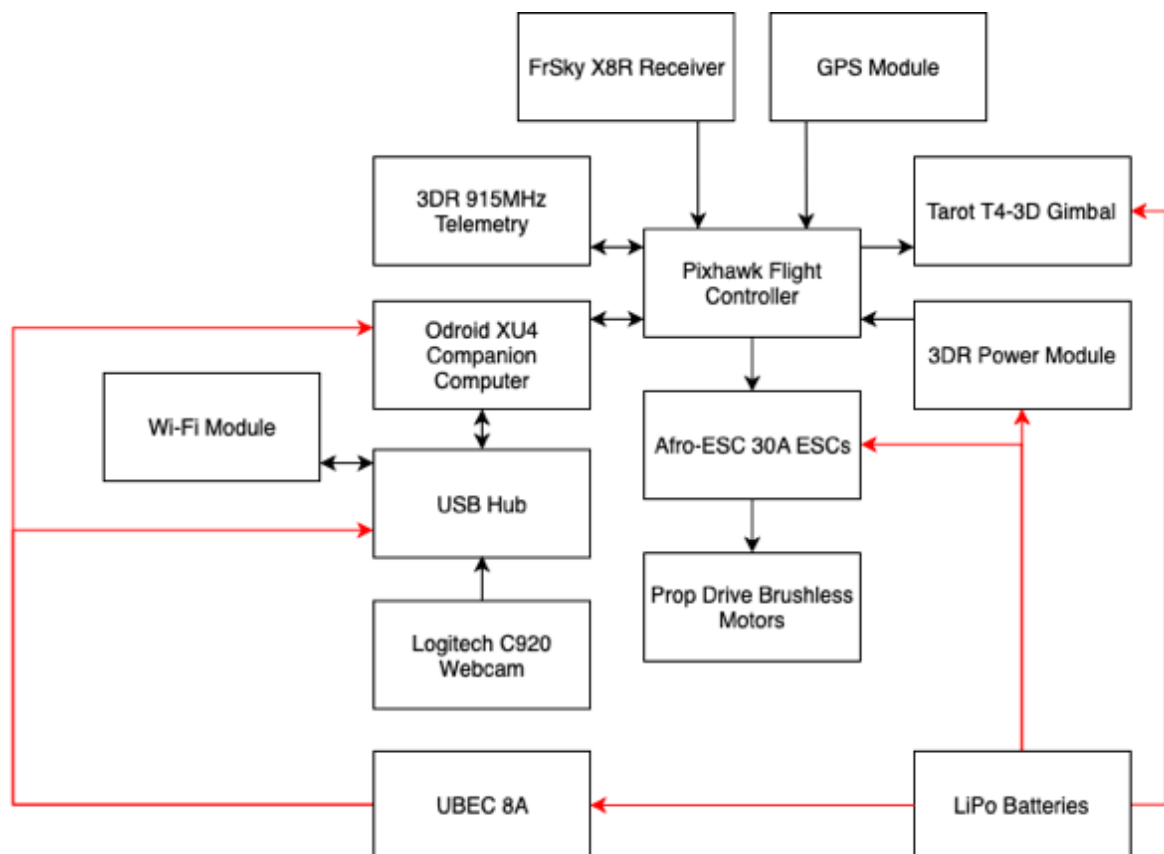


Figure 3.28: Block diagram for the final hardware architecture of the aircraft

3.2.17 Pixhawk Configuration

There are a number of peripherals connected to the Pixhawk flight controller. To help explain the setup, the table below shows the devices that were connected to their specific pinout on the Pixhawk. The schematic and pinout for the Pixhawk can be found in Appendix 3.6.

Table 3.5: Peripherals connected to the Pixhawk

Device	Pixhawk Pinout
Telemetry (915 MHz)	TELEM 1
FTDI (Companion Computer)	TELEM 2
GPS Module (GPS + Compass)	GPS & I2C
Power Module	POWER
Receiver	RCIN
ESCs (x6)	MAIN OUT 1 – MAIN OUT 6
Gimbal (pitch and yaw axes)	AUX OUT 1 & AUX OUT 2

The connections between the telemetry, FTDI, GPS module and power module with their respective Pixhawk pinouts can be found in Figure 3.29.

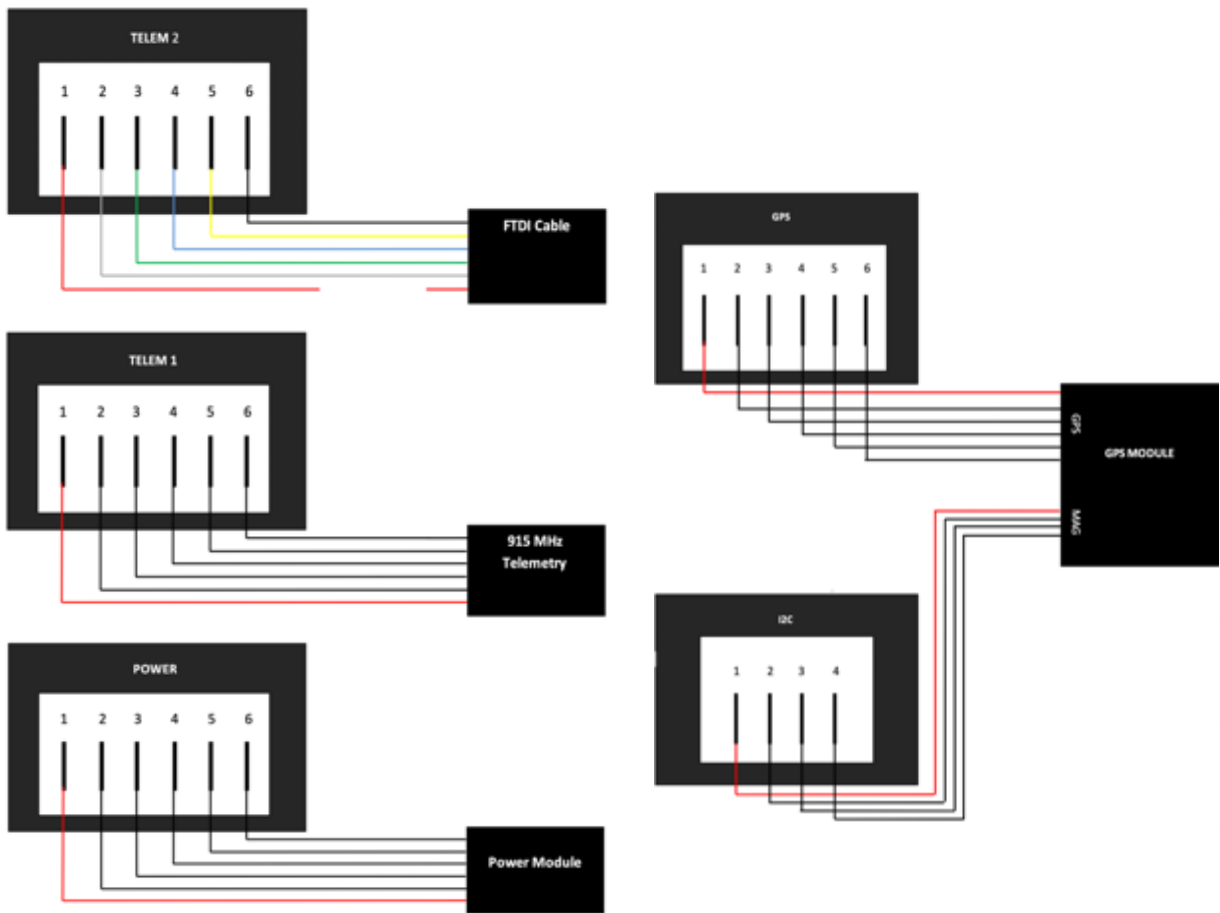


Figure 3.29: Peripherals connected to the Pixhawk

Referring to Figure 3.29, it can be noted that the FTDI cable contained colour coded cables within the cable. The colour of each wire and the pin it is connected to can be seen in Table 3.6. Referring to Figure 3.29, it can also be noted that the red cable was disconnected (as in not attached to the pin).

Table 3.6: Colour code of the wiring of the FTDI cable and their pinouts

Colour	Pinout
Red	VCC
White	TX
Green	RX
Blue	CTS
Yellow	RTS
Black	GND

The connections between the receiver, the six ESCs and the gimbal with their respective pinouts can be found in Figure 3.30. The design for Figure 3.29 and Figure 3.30 was based off a similar design developed by Mr James Sewell (Sewell, 2019).

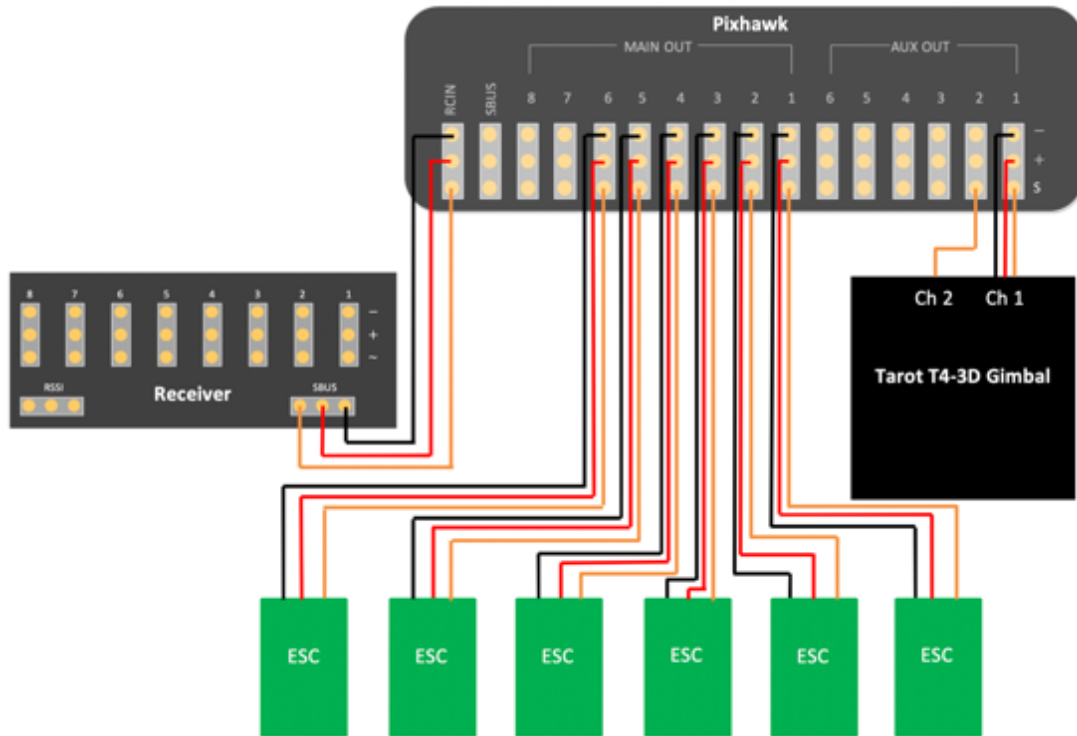


Figure 3.30: Receiver, gimbal and six ESCs connected to the Pixhawk

3.3 Software Architecture

This section will describe all the software programs that were used throughout the research as well as discuss the firmware that was used on the flight controller.

3.3.1 Autopilot Software

As discussed in section 2.4, an autopilot software needs to be installed onto the Pixhawk flight controller. Initially, when deciding which firmware would be used for this research, the main focus was to use the most stable software currently available. At the start of the research, the most commonly used software available was the one developed by ArduPilot, of which there are different versions available. The specific one that was used and installed onto the Pixhawk was the ArduCopter firmware as this firmware is specifically designed for multirotor aircrafts. Since there are stable

and beta versions of ArduCopter available, only the latest stable versions were ever uploaded onto the Pixhawk.

However, when the decision was made to use ROS for this research project, the Pixhawk's firmware needed to be switched to a more ROS compatible firmware. The decision was made to switch to PX4 firmware, where the most stable version was always used. Even though ArduPilot was compatible with ROS, there were still a few limitations to the way the firmware was integrable with ROS.

3.3.2 Ground Control Station Software

As discussed in section 2.5, the ground control station (GCS) is the software that is installed onto a ground-based computer, used to communicate with a UAV. It was decided that QGroundControl would be used as the main GCS software for this research as it supports both ArduPilot and PX4 firmware.

Before the aircraft could be taken for a test flight, the following needed to be set up via QGroundControl to use ArduCopter on the aircraft:

- Uploading the ArduCopter firmware onto the Pixhawk.
- Calibrating the Pixhawk's compass.
- Calibrating the Pixhawk's gyroscope.
- Calibrating the Pixhawk's accelerometer.
- Calibrating the Pixhawk's level horizon.
- Calibrating the transmitter and assigning its channels to the Pixhawk (e.g. channel 6 on the transmitter will arm the aircraft).
- Selecting which flight modes to be available for use by the aircraft.

The above were also needed to be set up for the Pixhawk when running PX4 firmware. The following additional steps were required for setup:

- Maximum voltage per battery cell.
- Minimum voltage per battery cell.

ArduPilot and PX4 offer numerous flight modes to be used on a multirotor aircraft. However, only 6 were selected for use with the ArduPilot firmware and 7 were selected

for use with the PX4 firmware. Table 3.7 and Table 3.8 show the flight modes that were selected for each firmware, as well as a description of the flight mode.

Table 3.7: Description of the flight modes used on the aircraft while using ArduPilot firmware (ArduPilot, 2019)

Flight mode	Description of flight mode
Acro	Holds its attitude, but has no self-levelling.
Stabilize	Self-levels itself along the roll and pitch axis.
PosHold	Holds its altitude and position by using the GPS. However, it has manual roll and pitch when the transmitter sticks are not centred.
Loiter	Holds its altitude and position by using the GPS.
Auto	Performs a pre-defined autonomous mission.
RTL (return to land)	Returns and lands at the take-off location.

Table 3.8: Description of the flight modes used on the aircraft while using PX4 firmware (PX4, 2019c)

Flight Mode	Description of flight mode
Acro	Holds its attitude, but has no self-levelling.
Stabilized	Self-levels itself along the roll and pitch axis.
Position	Holds its altitude and position by using the GPS. However, it has manual roll and pitch when the transmitter sticks are not centred.
Mission	Executes a pre-defined mission/flight plan.
Return	Aircraft ascends to a safe height and returns to its home position, where it will land.
Land	Lands at the current location.
Offboard	Obeys a position, velocity or attitude setpoint provided over MAVLink (often from a companion computer).

Other GCS software that was used to some extent were Mission Planner and MavProxy. Mission Planner is a GCS software similar to QGroundControl. However, the software is limited to only ArduPilot. Also, the graphical user interface (GUI) of the

software is not as user friendly as QGroundControl. MavProxy is a command line ground control station. However it was barely used due to its having a limited GUI interface.

3.3.3 PX4 Parameters

There were numerous parameters that needed to be adjusted in order to allow the aircraft to perform autonomous landing by making use of the companion computer. The parameters that were adjusted on the PX4 firmware on the flight controller can be referred to in Appendix 3.7.

3.3.4 Companion Computer Software

The following operating system and software were installed and used on the companion computer.

3.3.4.1 Operating System

As discussed in section 3.2.10, an eMMC was purchased as the storage device for the Odroid XU4. There were two choices between which operating system could be used on the Odroid. They were Android and Linux. The decision was made to opt for Linux as ROS only works on Linux. The version of Linux that came on the eMMC was Ubuntu Mate, which is a less processor demanding operating system than the original Linux Ubuntu. On the arrival of the computer and the eMMC, the operating system was immediately upgraded to the latest version available, which at the time was Ubuntu Mate 16.04.3-4.14. This was done by downloading the latest image file (.iso file) from Hardkernel's website. The image was flashed to the eMMC module by using Etcher software.

3.3.4.2 ROS

Once the operating system was installed, robot operating system (ROS) was installed onto the Odroid. The ROS version that was installed and used throughout the research was ROS Kinetic. The installation was followed based on the installation guide available on the ROS website. In order to use ROS, a workspace was needed to be created to store all the packages, scripts of code and launch files that would be used. For this research project, the workspace that was developed was called

hexaircraft_ws and the package where all the scripts were developed in was called hexaircraft.

3.3.4.3 Terminator

When it came to developing the code, the fact that the terminals that were used were scattered around the computer screen, was frustrating. To rectify this, a program called Terminator was installed. This helped group the terminals neatly as well as provide for a layout to be set up, allowing for multiple terminals to be opened and perform a unique task, all via the execution of one command in a terminal. This helped save a lot of time when having to restart and open the terminals when developing the code. An example of Terminator being used as well as its layout can be found in Figure 3.31.

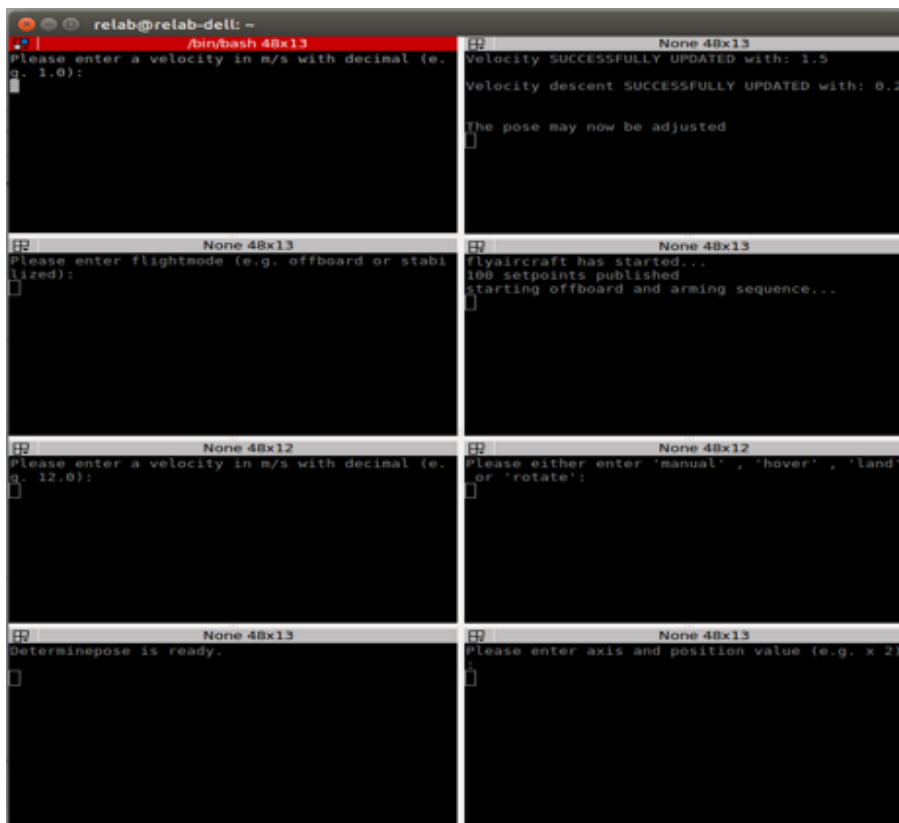


Figure 3.31 Terminator used to group terminals

3.3.5 Tarot Gimbal Software

Both Tarot gimbals that were used offer software available to allow parameters to be adjusted and calibrations to be made on the gimbal. Figure 3.32 shows a screenshot

of the software that was used, where the parameters that were set for the gimbal can be seen.

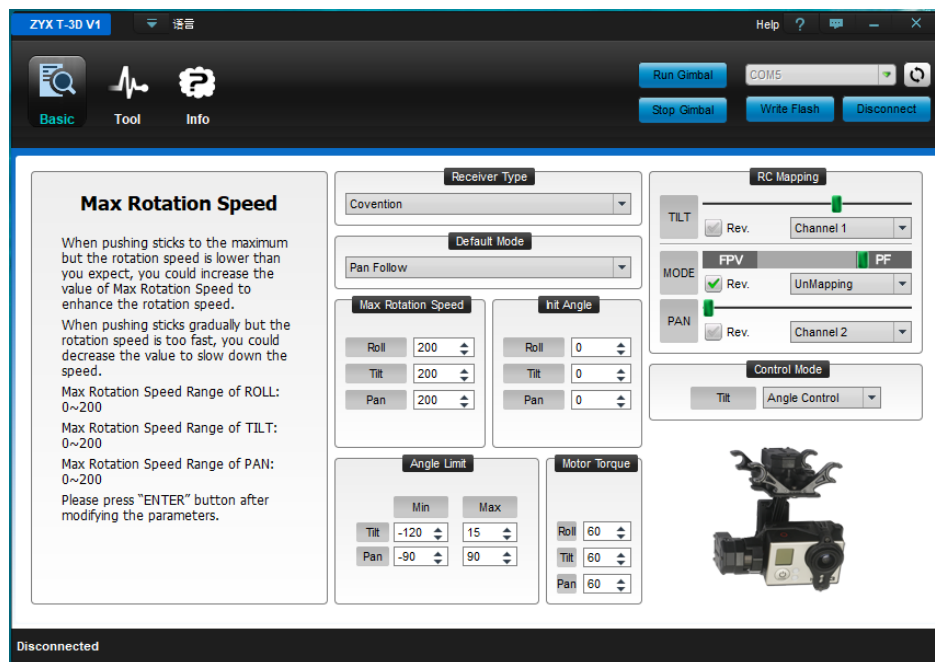


Figure 3.32: Tarot Gimbal Software

3.3.6 Virtual Machine Software

In order to write scripts and develop code for the companion computer to use, it was decided that the code would first be developed, either on a laptop or PC, then transferred across to the companion computer. This would allow for the code to be tested on a simulator before testing it on the actual aircraft (refer to chapter 6 to see more information on the simulation). To be able to write the code on the laptop (which was a Windows operating system laptop), a virtual machine was used to run Linux Ubuntu in parallel with the Windows operating system, allowing for both systems on the laptop to be used simultaneously. The only downside is that a virtual machine is not granted as much processing power as compared to the Windows system. The virtual machine software that was used was Oracle VirtualBox, which was downloaded for free from their website. An extension pack was also downloaded off their website, which supported extra features such as having support for USB 2.0 and USB 3.0 devices.

The virtual machine was also used to run ROS on the ground station computer to send commands through to the companion computer when a task needed to be performed. This was done by using the network that was established by the D-Link router.

3.3.7 Virtual Network Computing Software

Initially during testing of the actual aircraft on the field, a computer monitor, HDMI cable, keyboard and a mouse had to be taken to the field to set up the computer as well as to run all the necessary programs and scripts to use ROS alongside the aircraft. This became a tedious process as well as energy consuming (having to run back and forth between the aircraft and the ground station computer) to get the complete system up and running. To solve this problem, a VNC system was setup over the network. Virtual network computing (VNC) is a graphical desktop sharing system that allows a person to control the desktop interface of one computer from another computer or mobile device remotely (Raspberry Pi, 2017). The computer that is required to be remotely controlled needs to be running a VNC server whilst the computer that is going to be used to control the other computer remotely needs to be running a VNC viewer. Therefore, a VNC server needed to be set up on the companion computer and a VNC viewer needed to be set up on the ground station computer (as in the laptop). The software installed on the companion computer was called X11VNC Server and the software installed on the ground station computer was VNC Viewer. The VNC Viewer that was used in this research can be found in Figure 3.33.



Figure 3.33: VNC Viewer running on a MacBook Pro laptop

3.4 Conclusion

In this chapter, the dynamics of the aircraft were explained and the components used for the research and the integration of the hardware discussed. The setup of the flight controller was explained, shedding light on how the flight controller was connected to its peripherals as well as how the software was set up. Other software that was used throughout the research was discussed, emphasising its relevance. The next chapter, Vision System, will discuss the importance of the system and how it was established.

Chapter 4: Vision System

From the beginning of this research, it became evident that the most important part of this research would be the vision system. This chapter will be divided into three sections. The first will discuss the reason for using Robot Operating System (ROS) for the vision system; the second will deal with the use of the vision system to help land the aircraft; and finally the third will explain how the vision system is used for human detection.

4.1 Using Robot Operating System

One of the main factors that needed to be addressed was how the vision system would be developed due to the numerous software and programming libraries available on the internet. However, there were two approaches that stood out from the rest. The first approach was to integrate OpenCV libraries into the vision system and then command the aircraft to move to a new position via MAVLink. The second approach, which was suggested by Prof. Riaan Stopforth, was to use ROS to integrate the whole system, where the vision system would be developed within the system.

The following tables show the advantages and disadvantages to both approaches.

Table 4.1: Using OpenCV for the vision system

Advantages	Disadvantages
OpenCV is open source (free to use).	OpenCV is complicated to use.
OpenCV is fast due to it predominantly written in C/C++ .	Not a seamless integration with the aircraft.
OpenCV library has over 2500 optimized algorithms available to help with practically any vision detection scenario (OpenCV, 2019).	
Has a community of 47 000 people to help with issues (OpenCV, 2019).	

Table 4.2: Using ROS to develop the entire system

Advantages	Disadvantages
ROS is open source (free to use).	ROS is initially complicated to use.
Allows for the complete system to be integrated seamlessly (including the aircraft).	The only official supported languages are Python, C++ and Lisp (with Java and Lua currently being experimented).
Has over 3000 packages available to be used within a project (e.g. OpenCV can be used within ROS) (ROS, 2019c).	
ROS provides great tutorials on its website to learn to use the system.	
Great community available to help with solving issues.	

The decision was made to use ROS for this research mainly due to the seamless integration between all the major components of the system.

As discussed in section 3.3.4.2, the version of ROS used for this research was ROS Kinetic. Table 4.3 contains a list of the main ROS packages that were used in the ROS build, the author of the package as well as a description of the package.

Table 4.3: ROS packages used in the build

ROS Package	Author	Description of the Package
mavros	Vladimir Ermakov	Communication driver for various autopilots with MAVLink communication protocol. Used to translate code developed in ROS into MAVLink commands for the aircraft to understand and follow (ROS, 2018c).
aruco_ros	Rafael Muñoz Salinas, Bence Magyar	Provides real-time marker based 3D pose estimation using ArUco markers (ROS, 2014).
darknet_ros	Marko Bjelonic	Real-time object detection system (ROS, 2018a).
usb_cam	Benjamin Pitzer	ROS driver for V4L USB cameras (ROS, 2016a).
camera_calibration	James Bowman, Patrick Mihelich	Allows for the calibration of monocular or stereo cameras using a checkerboard calibration target (ROS, 2017).
rqt	Dirk Thomas, Dorian Scholz, Aaron Blasdel	Framework for GUI development for ROS (ROS, 2016b).
video_stream_opencv	Sammy Pfeiffer	Contains a node to publish a video stream (e.g. from a pre-recorded file) (ROS, 2018b).

4.2 Camera Calibration

In order to use a camera in ROS, the first step to perform is to calibrate the camera. As discussed in section 3.2.11, the Logitech C920 webcam was purchased to be used on the aircraft. Therefore, the webcam was used in ROS. A camera calibration guide, developed by Robotics with ROS, was followed to help with this calibration process (Robotics with ROS, 2017).

The first step was to disable the autofocus of the camera. This was to prevent the camera from auto adjusting itself during the calibration process as well as when the camera was being used for the vision system. This was done by installing software on Ubuntu called `uvcdynctrl` and disabling the autofocus within the software.

The second step was to run the `usb_cam` package to connect the webcam to ROS. The camera in ROS was called:

```
/usb_cam
```

The video feed in ROS was published to the topic:

```
/usb_cam/image_raw
```

To view the video feed, the program `rqt` was used, which was a GUI development for ROS.

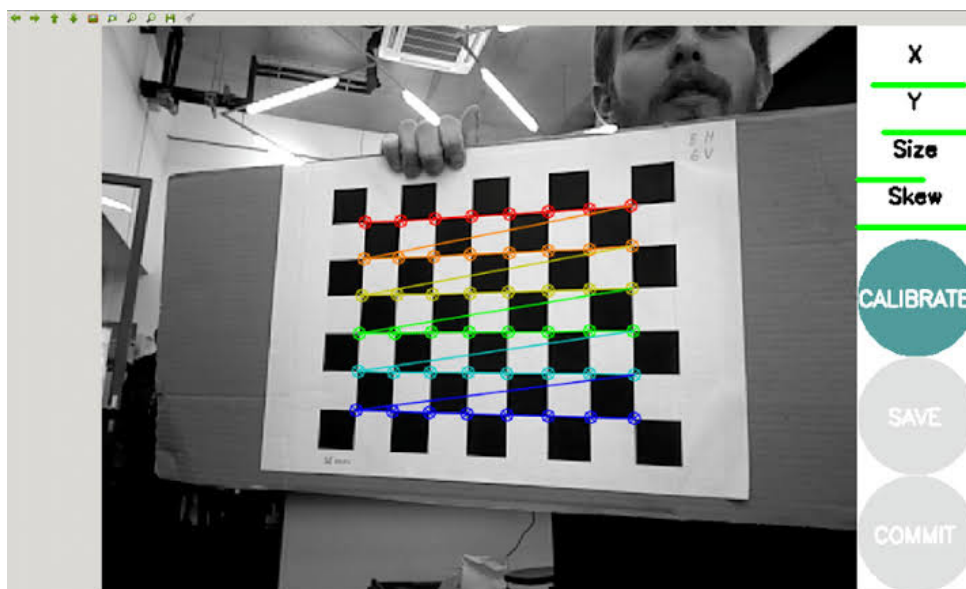


Figure 4.1: Camera calibration being performed using the checkerboard (Shipitko, 2017)

To perform the calibration, the ROS package `camera_calibration` was used, where a checkerboard calibration target was shown to the camera. The checkerboard used can be found in Appendix 4.1. The checkerboard contained black and white squares, where the camera would measure the distance between the inner corners of the squares. An example of this calibration being implemented can be seen in Figure 4.1. The size of the checkerboard was 8x6 (which means the number of inner corners was

8x6) and the size of each square was 25.7 mm. The checkerboard was printed on an A4 sheet of paper. The checkerboard was then:

1. Moved in front of the camera in a vertical and horizontal position.
2. Moved closer and further away from the camera.
3. Tilted (skewed) in different directions towards the camera.

Once the calibration was complete, the calibration data was saved in a .YAML file and placed in the ROS directory on the computer, where it could be used by any ROS package that wishes to use the calibration data.

4.3 ArUco Marker Detection

One of the main objectives for this research was to develop a vision system to help with the landing of the multirotor aircraft. It was decided that fiducial markers, specifically ArUco markers, would be used to be detected by the vision system. This was decided because the ArUco marker was able to provide a positional and orientational estimation (pose estimation) based on where the ArUco marker was positioned in relation to the camera. This would help the aircraft to align itself with the marker and then land on the marker, resulting in a potential accurate landing system.

To use ArUco markers with this system and specifically in ROS, the package `aruco_ros` was used (where the original ArUco dictionary was used within the package). This package was downloaded from the GitHub page assigned for `aruco_ros`. The package works by taking the topic (`image:=/usb_cam/image_raw`) published by the `usb_cam` package and then perform a check for any ArUco markers that have the same ID as the ID that had been pre-set by the `aruco_ros` package. For example, if an ID of 5 had been set in the `aruco_ros` package, the package would only search in the video feed for an ArUco marker that had an ID of 5. For this entire research, an ArUco marker with an ID of 7 was used. This specific marker can be seen in Figure 4.2. The `aruco_ros` package was run by using the launch file available in its package.

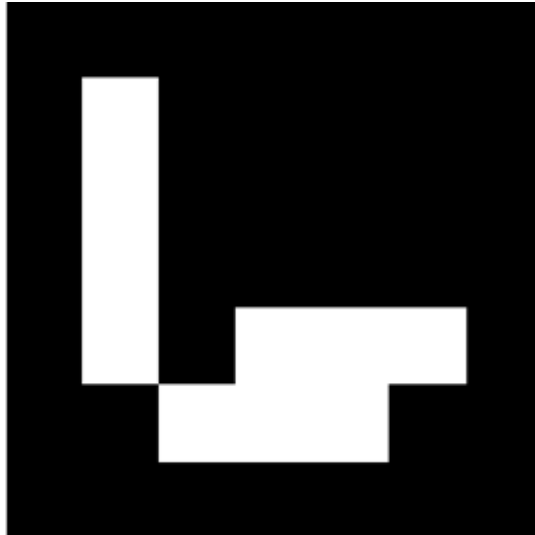


Figure 4.2: ArUco marker with an ID of 7

If the `aruco_ros` had detected the marker with the same ID as the one requested, the package had a topic that it would publish to, where the detected marker as well as the marker's axes (x, y and z) were shown in the video feed. The video feed (which was viewed using the `rqt` package) for this published topic was called:

`/aruco_single/result`

The positional and orientational estimation from the marker was published to a separate topic, called:

`/aruco_single/pose`

This topic could be broken into two sections (with a total of 7 values outputted to the topic):

1. Positional estimation: in the form of a co-ordinate
 - X co-ordinate
 - Y co-ordinate
 - Z co-ordinate
2. Orientational estimation: in the form of a quaternion
 - W – scalar number
 - X – imaginary number
 - Y – imaginary number
 - Z – imaginary number

This topic was available to be used and integrated into a node that could be self-developed. A node called `determinepose.cpp` was developed, where this topic was subscribed to (more information on this in section 5.1.3.1).

4.4 Human Detection

This section will be broken into four sections:

1. Discussing the use of the Darknet: YOLO package with ROS.
2. Calculating the ground distance between the aircraft and the detected person.
3. Calculating the angle to centre the person in the frame.
4. Calculating the new coordinates for the aircraft to fly towards.

4.4.1 Using Darknet: YOLO with ROS

As discussed in section 2.10.2, Darknet's YOLO is an open source neural network used for object detection. A ROS package called `darknet_ros` was developed, which allows YOLO to be integrated into a ROS project. As discussed in section 2.10.2, YOLO, based on the COCO dataset, is able to detect 80 objects, where one of the objects that it is able to detect is a person. This package was downloaded from the GitHub page that is assigned for `darknet_ros`. It must be noted that the Odroid XU4 companion computer was not capable of running Darknet's YOLO due to the insufficient processing power available. As a result, all the human detection aspects of this research were performed in a simulation on a desktop computer (more information on this in section 6.3).

In order to use the `darknet_ros` package, the first step was to choose which version of YOLO would be used for the vision system. The version of YOLO used for this package was tiny YOLO version 3 (referred to as YOLOv3-tiny). In order to use this version, the following two files were downloaded from Darknet YOLO's website:

1. `yolov3-tiny-voc.cfg` (the file stores the configuration for the neural network)
2. `yolov3-tiny.weights` (the file contains the trained weights for the neural network)

Another file, `yolov3.yaml`, was duplicated and renamed to `yolov3-tiny.yaml` (this file contains the objects/classes that can be detected as well as the detection probability threshold).

In order to use the Logitech C920 webcam to detect objects using `darknet_ros`, a file in the package needed to be amended. The `ros.yaml` file was configured to use the topic that would be published by the `usb_cam` package. This topic was called:

```
/usb_cam/image_raw
```

If a pre-recorded video file was wanted to be run through `darknet_ros` instead of using a webcam, the above topic would need to be replaced with the topic (being published) that contained the pre-recorded file. In order to do this, a package called `video_stream_opencv` was used. This package was used when performing the testing for human detection (refer to section 6.3). The topic that was published by this package, containing the pre-recorded video file, was:

```
/videofile/image_raw
```

Once the launch file had been run, the video feed was searched for any possible objects. If the ROS package detected a possible object, it would give a probability that the detected object was correct. If the probability of the correct type of object detected was higher than the probability threshold that was set in the `yolov3-tiny.yaml` file, information based on the object would be published to a topic called:

```
/darknet_ros/bounding_boxes
```

A bounding box would also appear around the detected object in the video feed (where this was published to a separate topic). An example of a bounding box around a detected object can be seen in Figure 4.3.



Figure 4.3: Person detected on a field, where a bounding box was estimated

The following information was published to the `/darknet_ros/bounding_boxes` topic, if an object was detected:

1. Class
2. Probability
3. Xmax
4. Xmin
5. Ymax
6. Ymin

The bounding box was based on the resolution of the video feed, where it gave the position at a specific pixel. For example, if a video feed had a resolution of 1920x1080 (Full HD), the maximum Xmax value possible would be 1920 and the maximum Ymax value possible would be 1080. All bounding boxes were based on the origin being in top left corner of the screen.

The topic containing the bounding box information was available to be used and integrated into a node that could be self-developed. A node called `boxinfo.cpp` was developed, where this topic was subscribed to (more information on this in section 5.2.2.2).

4.4.2 Calculating the ground distance

If the idea is to develop a multirotor aircraft that can detect and follow a person, some necessary calculations will be required. One of the first calculations is to calculate the ground distance that the aircraft is away from the person.

Below is an example, where the aircraft is at an altitude of 10 metres and the person's bounding box (detected by darknet_ros) has a Ymax of 380 pixels .

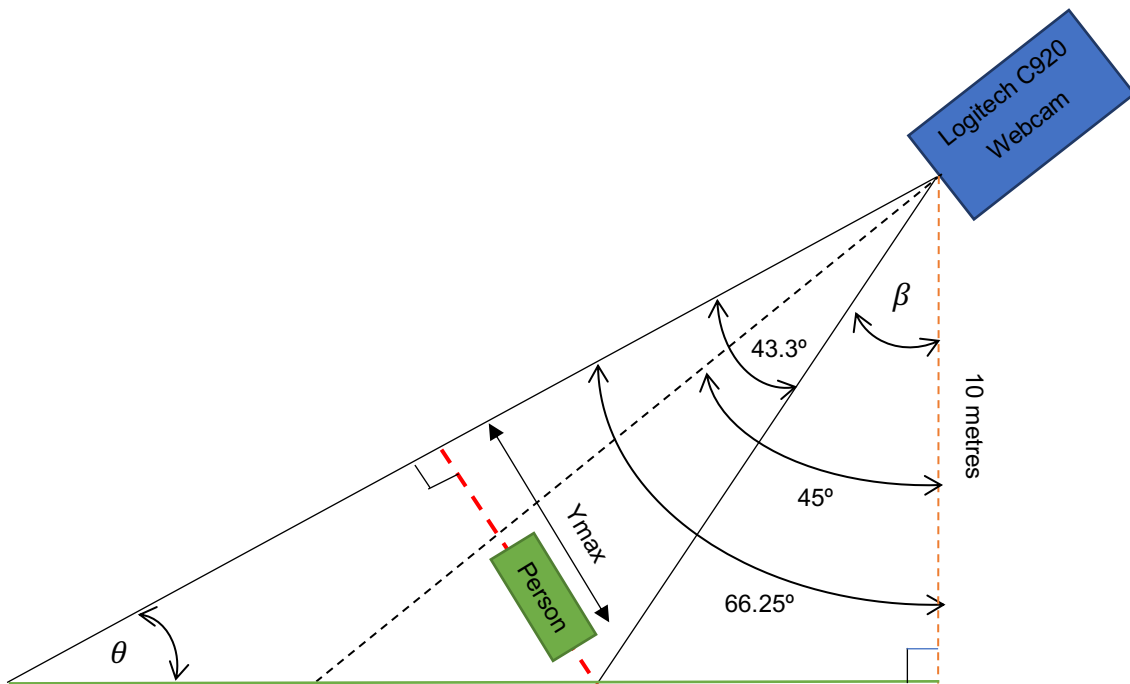


Figure 4.4: Diagram demonstrating the frame seen by the camera

The red dotted line represents the frame that is seen by the camera. The green box represents the bounding box that formed around the person, determined by using darknet_ros. The reason that the green box appears to be floating is because it resembles the person not being exactly at the bottom of the frame but actually standing at some distance away from the camera, resulting in the person being higher up in the frame. The webcam will be set at 45° by adjusting the angle of the gimbal that is holding the camera. The orange dotted line resembles the altitude that the aircraft is above the ground. For demonstration purposes, the altitude will be set at 10 metres.

To perform the calculations, the camera's field of view was required. The webcam had the following specifications (Logitech Apps, 2019):

Diagonal field of view (FOV) = 78°

Horizontal FOV = 70.42°

Vertical FOV = 43.3°

Focal length = 3.67mm

The resolution of the video feed was set at 640 x 480 pixels. This was because the Odroid XU4 companion computer struggled to handle the video processing of the camera's capable 1920 x 1080 (Full HD) resolution. However, the actual test was only performed on the simulation and not on the Odroid.

The ground distance calculation will be broken into two sections:

1. Calculating the distance of the aircraft from the bottom of the video frame.
2. Calculating the distance of the person within the video frame.

4.4.2.1 Calculating the distance of the aircraft from the bottom of the video frame

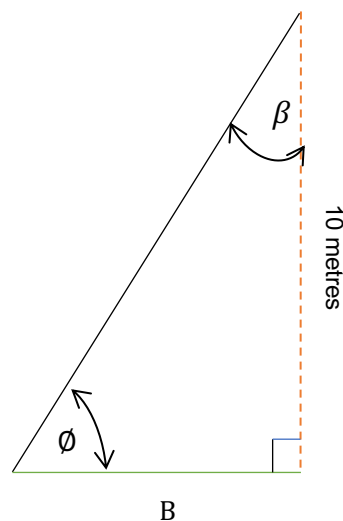


Figure 4.5: Angle between the aircraft and the bottom of the camera's field of view

$$\beta = 45^\circ - \frac{43.3^\circ}{2}$$

$$\beta = 23.35^\circ$$

$$\phi = 180^\circ - 90^\circ - 23.35^\circ$$

$$\phi = 66.65^\circ$$

$$B = \tan(\beta) \times 10$$

$$B = \tan(23.35^\circ) \times 10$$

$$B = 4.32 \text{ metres}$$

4.4.2.2 Calculating the distance of the person within the video frame

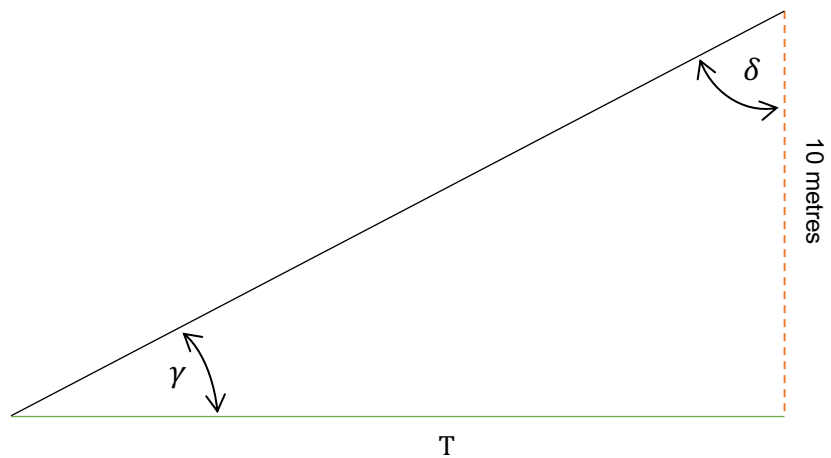


Figure 4.6: Angle between the aircraft and the top of the camera's field of view

$$\delta = 45^\circ + \frac{43.3^\circ}{2}$$

$$\delta = 66.65^\circ$$

$$T = \tan(\beta) \times 10$$

$$T = \tan(66.65^\circ) \times 10$$

$$T = 23.16 \text{ metres}$$

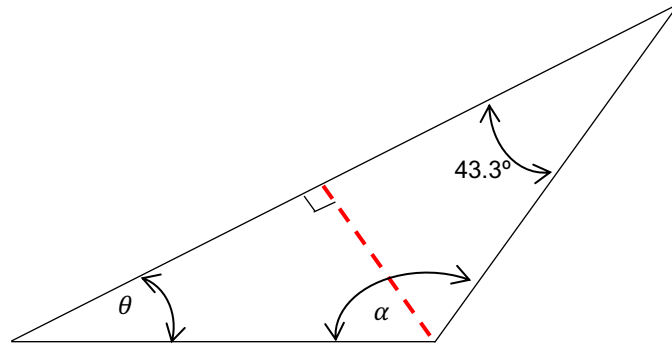


Figure 4.7: Angles within the camera's field of view

$$\alpha = 180^\circ - \phi$$

$$\alpha = 180^\circ - 66.65^\circ$$

$$\alpha = 113.35^\circ$$

$$\theta = 180^\circ - 113.35^\circ - 43.3^\circ$$

$$\theta = 23.35^\circ$$

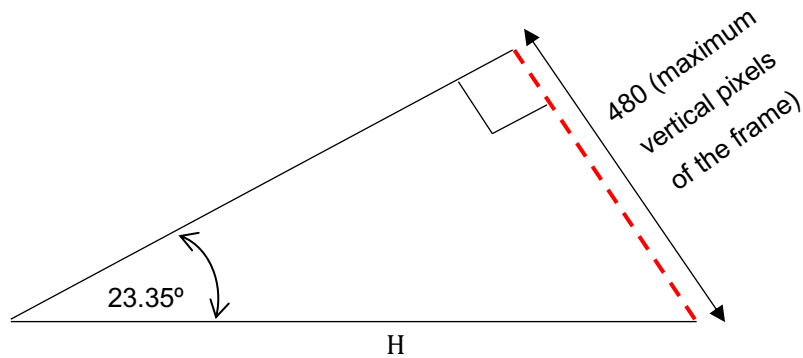


Figure 4.8: Triangle formed within the camera's field of view

$$H = \frac{\text{maximum vertical pixels}}{\sin(\theta)} \quad (4.1)$$

$$H = \frac{480}{\sin(23.35^\circ)}$$

$$H = 1211.06 \cong 1211 \text{ pixels}$$

To determine the distance with respect to only the video frame, the Ymax from the bounding box of the person needs to be used.

$$\text{Pixels from the bottom of the frame} = \text{Total vertical pixels} - Y_{\max} \quad (4.2)$$

$$\text{Pixels from the bottom of the frame} = 480 - 380$$

$$\text{Pixels from the bottom of the frame} = 100 \text{ pixels}$$

The distance of the person in the frame can be calculated by using linearization. This is done by comparing the size of the two triangles, seen in Figure 4.9

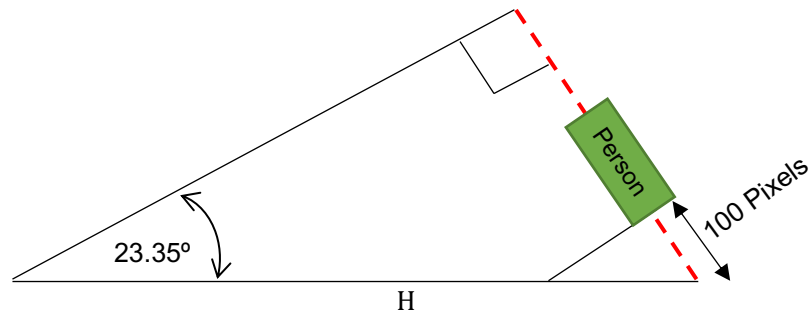


Figure 4.9: Same triangle as in figure .

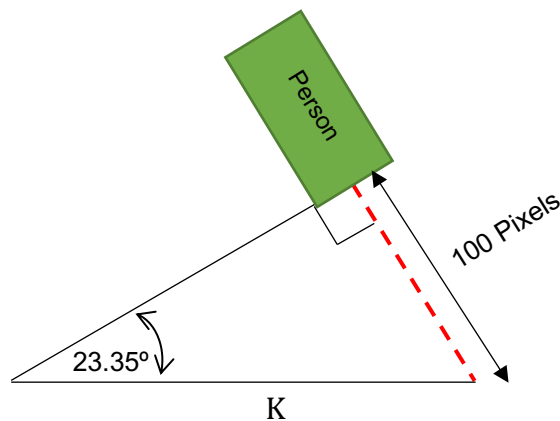


Figure 4.10: Smaller triangle formed underneath the bounding box of the person

$$K = \frac{100}{\sin(23.35^\circ)} \quad (4.3)$$

$$K = 252.30 \cong 252 \text{ pixels}$$

By linearization:

$$\frac{252.30}{1211.06} = \frac{x}{(T-B)} \quad (4.4)$$

$$x = \frac{252.3}{1211.06} \times (23.16 - 4.32)$$

$$x = 3.92 \text{ metres}$$

Therefore, the total ground distance between the aircraft and the person is:

$$\text{total ground distance} = x + B \quad (4.5)$$

$$\text{total ground distance} = 3.92 + 4.32$$

$$\text{total ground distance} = 8.24 \text{ metres}$$

It must be noted that the aircraft should only move forward by the calculated distance of the person in the video frame (in the case of the above example, 3.92 metres). This is because you do not want the aircraft to fly to the exact point where the person is positioned or else the person will no longer be in the video feed. Therefore, throughout the rest of this dissertation, the ground distance will refer to the calculated distance of the person in the video frame.

The abovementioned calculations will be implemented into a script to be used by the vision system. With regard to the script that will be developed, the altitude and the vertical resolution will need to be stored in separate variables, allowing them to be adjusted easily, if need be.

4.4.3 Centre a person in the frame

Figure 4.11 shows a top view of an aircraft, where it has detected a person and calculated the bounding box for the person. The idea is to calculate the angle the aircraft needs to rotate by making use of the bounding box, the horizontal resolution of the video feed and the horizontal field of view of the camera.

With regards to the bounding box, the midpoint will be used. This can be calculated by:

$$\text{midpoint of bounding box} = \frac{X_{max} + X_{min}}{2} \quad (4.6)$$

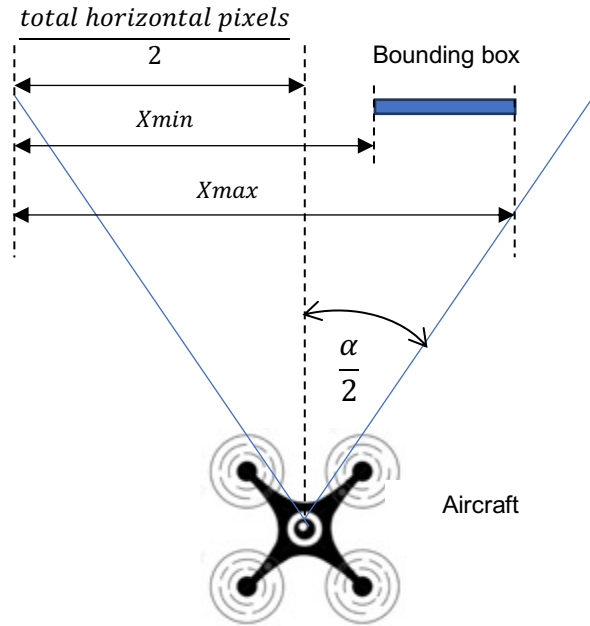


Figure 4.11: Top view of the aircraft detecting a person

To determine whether the aircraft needs to rotate left or right:

$$Rotate = \frac{\text{total horizontal pixels}}{2} - \text{midpoint of bounding box} \quad (4.7)$$

If *Rotate* is negative, the aircraft must rotate right and if its positive or equal to zero, the aircraft must rotate left.

Therefore, the angle to rotate the aircraft has two formulas depending on the direction the aircraft needs to rotate:

1. Aircraft needs to rotate right:

$$\begin{aligned} \text{Angle to rotate aircraft right} &= \frac{\text{Horizontal FOV}}{2} \times \frac{\frac{X_{max}+X_{min}}{2}}{\frac{\text{Horizontal Resolution}}{2}} \\ &= \frac{\text{Horizontal FOV}}{2} \times \frac{X_{max}+X_{min}}{\text{Horizontal Resolution}} \end{aligned} \quad (4.8)$$

2. Aircraft needs to rotate left:

$$\begin{aligned} \text{Angle to rotate aircraft left} &= \frac{\text{Horizontal FOV}}{2} \times \frac{\frac{\text{Horizontal Resolution} - (X_{max}+X_{min})}{2}}{\frac{\text{Horizontal Resolution}}{2}} \\ &= \frac{\text{Horizontal FOV}}{2} \times \frac{\text{Horizontal Resolution} - (X_{max}+X_{min})}{\text{Horizontal Resolution}} \end{aligned} \quad (4.9)$$

Equation 4.8 and Equation 4.9 will be applied to a script to perform the calculation.

4.4.4 Developing coordinates for the aircraft using the calculated ground distance and calculated angle

The idea was to develop a method that could calculate a coordinate for the aircraft to fly to, based on the bounding box that was formed using the `darknet_ros` package.

To calculate the new potential coordinate, the following will need to take place:

1. Use the calculated ground distance.
2. Use the calculated angle that the aircraft needs to rotate towards, as well as the angle that the aircraft is rotated by at the time of the calculation (in other words, the direction that the aircraft is facing).
3. The new coordinate will be calculated based on a cartesian coordinate system, where the aircraft's home position and current position will be used.

Before continuing with the calculation, the aircraft's frame of reference needs to be discussed. The aircraft's frame of reference has been established by PX4. PX4 uses a FRD (X Forward, Y Right and Z Down) for the local body frame and NED (X North, Y East and Z Down) for the local world frame. However, the ROS frame is different to the PX4 frame. Table 4.4 explains what frame is used for ROS and PX4 and Figure 4.12 demonstrates the frames used for ROS and PX4.

Table 4.4: Frame of reference for ROS and PX4

Frame	ROS	PX4
Body	FLU (X Forward, Y Left and Z Up)	FRD (X Forward, Y Right and Z Down)
World	ENU (X East, Y North and Z Up)	NED (X North, Y East and Z Down)

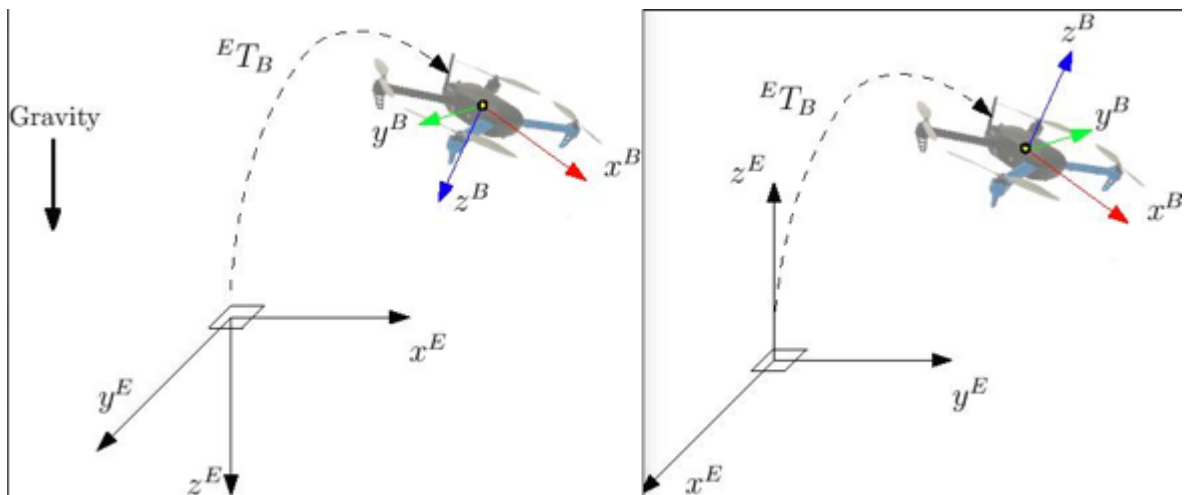


Figure 4.12: Frame of reference for ROS and PX4, where North East Down (NED) on the left and East North Up (ENU) on the right (PX4, 2019h)

The following is an example of how the new co-ordinate would be calculated if the new co-ordinate formed in the first quadrant relative to the aircraft. For this example, the aircraft is currently at the co-ordinate (-2;5). A distance of 17 metres will be used as the calculated ground distance for the example.

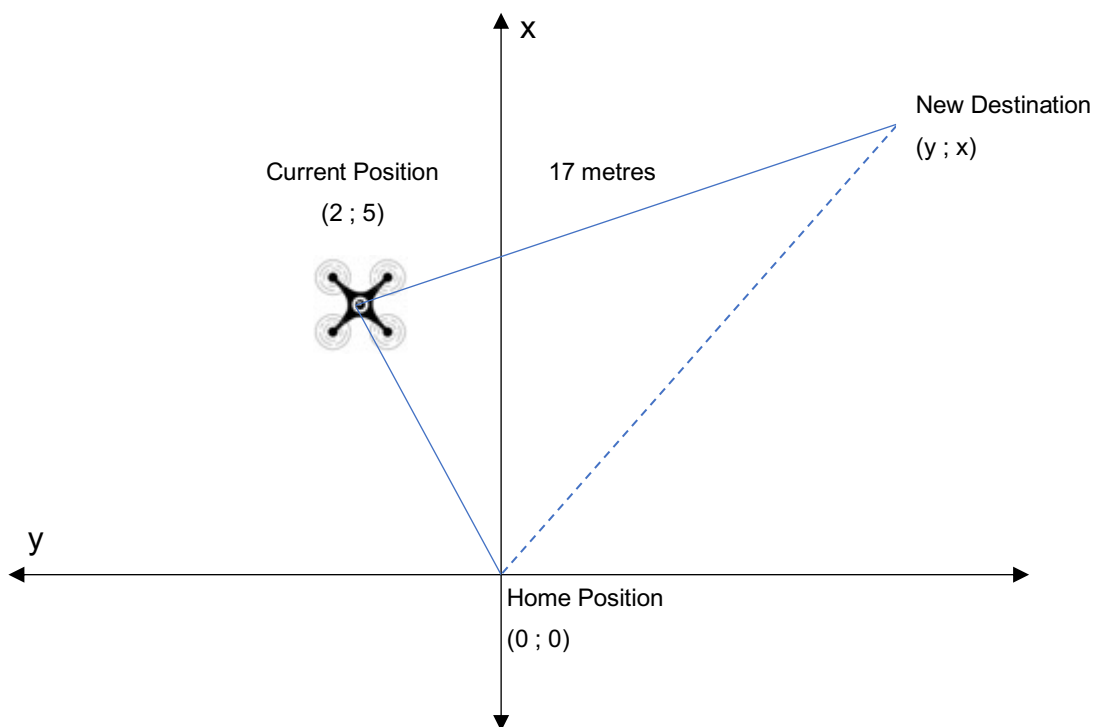


Figure 4.13: Example to demonstrate how the co-ordinate is calculated

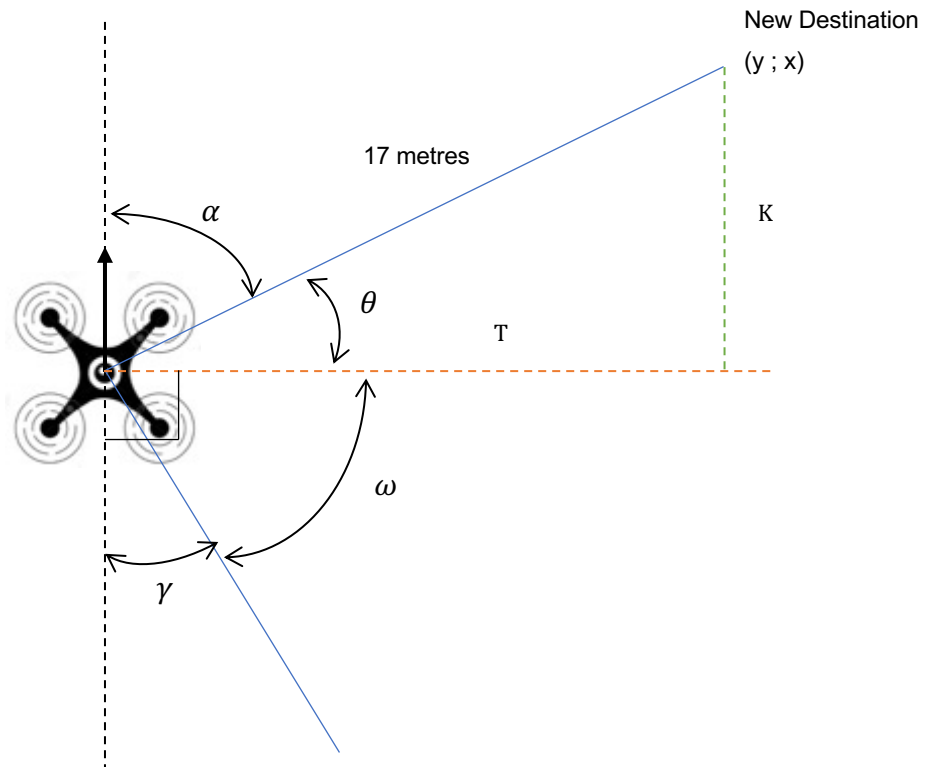


Figure 4.14: Angles around the aircraft used to calculate the new co-ordinate

The angle α is the angle that the aircraft is facing after it has rotated to centre the person in the middle of the frame. This angle can be determined by reading the quaternion from the aircraft, then converting the quaternion to an Euler angle to obtain the rotated angle about its z axis.

For now, α will be approximated at 70° for demonstration purposes.

$$\theta = 90^\circ - \alpha$$

$$\theta = 90^\circ - 70^\circ$$

$$\theta = 20^\circ$$

Thus, T and K can be calculated:

$$T = 17 \times \cos(20^\circ)$$

$$T = 15.97 \text{ metres}$$

$$K = 17 \times \sin(20^\circ)$$

$$K = 5.82 \text{ metres}$$

Therefore, the y and x co-ordinates are as follows:

$$y = \text{aircraft } y \text{ coordinate} - T \quad (4.10)$$

$$y = 2 - 15.97$$

$$y = -13.97$$

$$x = \text{aircraft } x \text{ coordinate} + K \quad (4.11)$$

$$x = 5 + 5.82$$

$$x = 10.82$$

This example was used to illustrate the principles that will apply. However, as mentioned, this example was based on the new co-ordinate being in the first quadrant (refer to Figure 4.14). Since there are four quadrants (based around the aircraft) that the new co-ordinate could be positioned at, the principles above would need to be adapted accordingly when developing the script to move the aircraft using ROS.

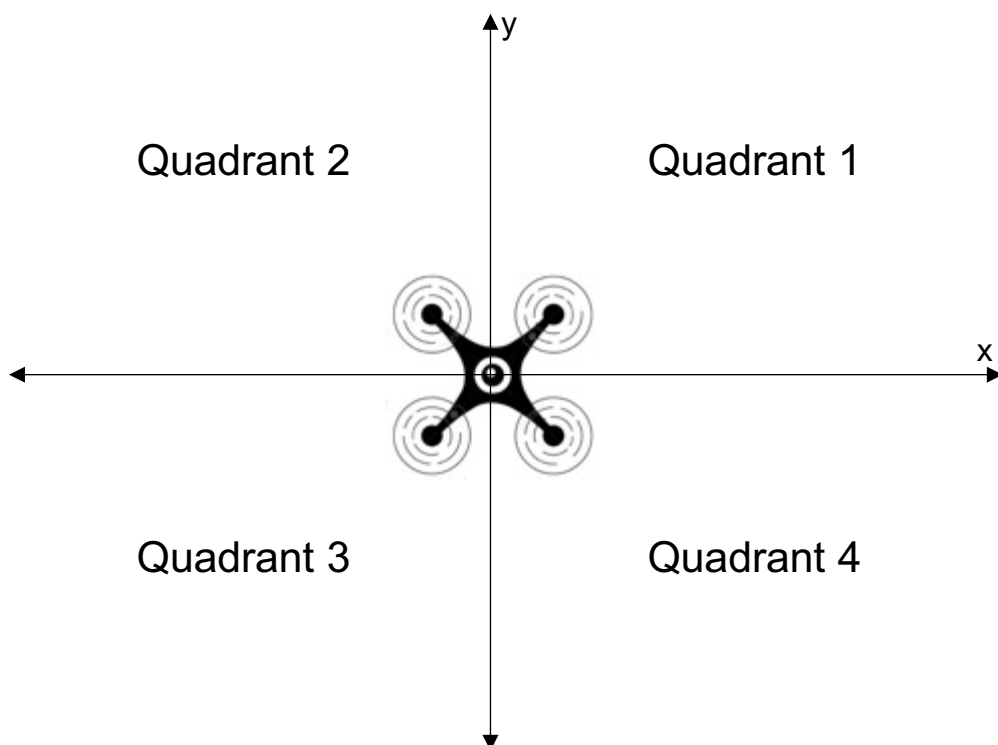


Figure 4.15: Quadrants around the aircraft

4.5 Conclusion

To conclude this chapter, the reasoning behind using ROS to integrate the whole system (including the vision system) was mentioned. The ROS packages used throughout the research were discussed in detail. A camera calibration was performed for the Logitech C920 webcam. The vision system for the landing of the aircraft and the vision system for the human detection system were discussed. Calculations for both the landing system and the human detection system were performed. The next chapter, Integrated System, discusses how the vision system was integrated with the aircraft by using ROS.

Chapter 5: Integrated System

The following chapter will explain how the entire system is integrated. Since ROS was used in the system, this chapter will mainly expand upon how ROS was used to integrate the system. Certain sections of code from the scripts that were developed and used in the system will be discussed in detail. The landing system and the human detection system will be discussed, where all the necessary calculations will be set out in detail. Flow charts will be included for the landing system and the human detection system. Finally, a user requirements specification for this system will be discussed.

5.1 The Landing System

This section will discuss why the landing system was developed, provide a basic overview of how the system works, as well as discuss the design of the system in detail, explaining all the scripts that were developed for the system. A flow chart will also be used to show how the system works.

5.1.1 Reasoning behind developing the landing system

Before discussing how the landing system was developed, the reasoning behind developing the system needs to be explained. Currently, the Pixhawk flight controller has the ability to be flown autonomously by making use of the GPS module mounted onto the aircraft. This GPS module would also be used to land the system autonomously. However, the GPS module, which contained a Neo-M8N GPS (as well as a compass) module, had the following statistics that were obtained from its datasheet (u-blox, 2015):

- Horizontal accuracy: 2.5 metres (autonomous)
- Heading accuracy: 0.3 degrees

This horizontal accuracy would not be sufficient if the aircraft was required to land in an accurate position (e.g. if a charging docking station is developed for the aircraft, where the docking station has a top plate of 1.5 x 1.5 metres).

There were other methods available at the time to help with the precision landing of an aircraft. These methods were:

1. Real Time Kinematic (RTK) GPS: A more advanced GPS system that attempts to eliminate the possible errors by making use of a second GPS receiver

positioned at the ground control station. The system is able to obtain centimetre-level accuracy (PX4, 2019d).

2. IR-Lock Sensor: This is a sensor (that is installed onto the bottom of the aircraft) that detects a MarkOne beacon that is positioned at the specified landing spot.

The system claims to obtain a precision landing of roughly 10cm (PX4, 2019e).

However, using one of these systems would result in additional expenditure in respect of the aircraft. Since a camera was already going to be installed onto the aircraft that would be used for human detection, the idea was to integrate a landing system into the entire system, making use of the camera.

5.1.2 Basic overview of the landing system

Before going into detail, a basic overview of the landing system is provided as follows:

1. The aircraft's flight mode is switched to Offboard mode via a node.
2. The aircraft will arm itself, take off and become airborne.
3. A node is run to inform ROS that the landing system is to commence.
4. The vision system will search for the ArUco marker positioned on the ground.
5. The system will reposition the aircraft over the ArUco marker, where the aircraft will hover directly above the marker.
6. Once the aircraft is above the ArUco marker, a timer will start, warning the ground control station that the aircraft is about to be rotated.
7. The system will begin to rotate the aircraft to align the aircraft with the orientation of the ArUco marker. The system will have a couple of seconds to rotate the aircraft.
8. After the time has elapsed, the system will cause the aircraft to descend, constantly keeping it positioned over the marker. If the aircraft drifts out of a horizontal tolerance range that has been set, the system will stop the aircraft from descending and force it to be repositioned directly above the marker.
9. The system will land the aircraft on top of the ArUco marker, where the aircraft will be automatically disarmed.

5.1.3 Design

There were two projects that assisted the development of the landing system:

1. PX4 had an offboard example available on their website, where a multirotor aircraft was able to take off and hover at an altitude of two metres. This project provided a platform to start from, where new code could be developed and implemented (PX4, 2019f).
2. Aerial Robotics had developed code where a multirotor aircraft was able to take off, detect an ArUco marker and hover above a ArUco marker, all being run in Gazebo simulation software. They also made use of PX4's offboard example. Aerial Robotics' ROS package could be found on their GitHub page (Aerial Robotics, 2019). This package helped demonstrate how an aircraft could be moved, based on detecting an ArUco marker.

These were the following nine ROS nodes that were developed for the landing system (where the nodes were written in C++ language):

1. `determinepose.cpp`: It's purpose is to determine the co-ordinates that the aircraft needs to fly towards. This is the script that performs all the necessary calculations to align the aircraft with the ArUco marker and land on the marker autonomously.
2. `flyaircraft.cpp`: Its main purpose is to communicate commands to and from the aircraft via MAVROS.
3. `talker_autoland.cpp`: It's purpose is to choose how the aircraft should be flown.
4. `talker_coordinates.cpp`: It allows the aircraft to be manually repositioned.
5. `talker_descent.cpp`: It changes the descending velocity of the aircraft.
6. `talker_velocity.cpp`: It changes the horizontal velocity of the aircraft.
7. `talker_flightmode.cpp`: It chooses the flight mode for the aircraft.
8. `talker_quat.cpp`: It allows the aircraft to be manually rotated by using a quaternion.
9. `listener_status.cpp`: This is the script used to receive all the commands that have been executed on the aircraft.

5.1.3.1 determinepose.cpp

The script was also used by the human detection system (more information on this in section 5.2.2.1). The following table is a list of subscribed topics (topics that provide information) that were contained in the node to be used in the landing system:

Table 5.1: The topics subscribed to in the determinepose.cpp node

Topic	Description of the topic
CoordinatesFromUser	Coordinates received from the talker_coordinates.cpp script.
QuatFromUser	Yaw the aircraft by sending a quaternion from the talker_quat.cpp script.
VelocityFromUser	Horizontal velocity received from the talker_velocity.cpp script.
DescentFromUser	Descending velocity received from the talker_descent.cpp script.
AutolandFromUser	How the aircraft should be flown, which is received from the talker_autoland.cpp script.
FlightmodeFromUser	Flight mode received from talker_flightmode.cpp script.
mavros/state	Receives the current state from the aircraft via MAVROS (e.g. whether the aircraft is armed or not).
mavros/imu/data	Receives the aircraft's IMU data via MAVROS in the form of a quaternion.
mavros/local_position/pose	Receives the aircraft's local position and orientation via MAVROS.
aruco_single/pose	Receives the ArUco marker's position and orientation relative to the aircraft's current position and orientation.
mavros/global_position/rel_alt	Receives the aircraft's current relative altitude, via MAVROS, from where the aircraft was armed.

The following table is a list of published topics (publishing topics for the other nodes to use) that were contained in the node to be used by the landing system.

Table 5.2: The topics published to in the determinepose.cpp node

Topic	Description of the topic
aircraftpose_pub	Publishes the determined position and orientation for the aircraft.
StatusFromAircraft	Publishes all the current commands that are taking place (e.g. when the aircraft is about to rotate).

In order for the aircraft to reposition itself above the ArUco marker, the following took place:

1. The positional information of the marker was used. Since the marker's frame of reference was different to the aircraft's frame of reference, the marker's positional information had to be converted in order for the aircraft to use it correctly. This is due to PX4's FRD frame (as discussed in section 4.4.4). The coordinate conversions implemented were:
 - aircraft's x coordinate = - (ArUco marker's y coordinate)
 - aircraft's y coordinate = - (ArUco marker's x coordinate)
 - aircraft's z coordinate = - (ArUco marker's z coordinate)
These coordinates were stored in a 1 x 3 matrix.
2. To stabilise the aircraft above the ArUco marker, the above calculated coordinates had to be multiplied by the aircraft's IMU. The IMU data, which was initially in the form of a quaternion, was converted into a rotational matrix (3 x 3 matrix) to be able to be multiplied to the co-ordinates matrix. This multiplication forms a 1 x 3 matrix, representing how far away the aircraft is from the ArUco marker in form of an x, y and z coordinate value.
3. The aircraft's current local position was added to the newly calculated x, y and z values. This step was crucial as it would allow the ArUco marker to be placed anywhere, irrespective of where the aircraft was armed. This will even allow the aircraft to follow the ArUco marker if it was mounted onto a moving object.

In order to rotate the aircraft so that it could properly aligned with the ArUco marker, the following took place:

- The aircraft first needed to be hovering above the ArUco marker. A tolerance was developed, allowing the aircraft to rotate even if the aircraft was not directly above the marker. This tolerance value was scaled, meaning that if the aircraft was further away from the marker, the tolerance range was greater and if the aircraft was closer to the marker, the tolerance range was smaller.
- Since the ArUco markers' frame of reference was different to the aircraft's frame of reference, the marker's orientation (as in quaternion) needed to be converted for the aircraft to use. To visualise the marker's quaternion and see what its respective Euler angles were, an online quaternion simulator was used. Screenshots from the online simulator can be found in Appendix 5.14. By using the simulator, a conversion method was determined by trial and error. The following was deduced:

- 1) Convert the quaternion to Euler angles.
- 2) Subtract the z axis component of the Euler angle by 180 degrees.
- 3) Invert the y axis component of the Euler angle.

For example, if the aircraft needed to rotate clockwise by an acute angle (smaller than 90 degrees) and the above three steps were followed, the outcome that would appear on the quaternion simulator can be found in Appendix 5.14.

However, a new method was discovered to convert the quaternion to be used by the aircraft, which can be found in Appendix 5.1 under the method called `ConvertArucoQuaternion`. This new method, which was also developed by trial and error, consisted of re-arranging the quaternion's components. The code developed for this re-arranging was designed to work only if the quaternion's components consisted of:

1. Both the x and y being a negative value.
2. Both the x and z being a negative value.

If the quaternion's components did not comply with the above two provisos, the quaternion was still able to be used. The four components just needed to be inverted, if the components looked as follows:

- Both the w and y being a negative value.
- Both the w and z being a negative value.

The following four tables show the re-arrangements of the components, depending on the angle by which the aircraft needed to be rotated. Only the w and y component values were used in the rearranging. For example, referring to Table 5.3, the new quaternion's z will equal the previous inverted w component.

Table 5.3: Quaternion formed if the aircraft needs to rotate clockwise ($0^\circ < \theta \leq 90^\circ$)

New Quaternion Components	Consists of previous Quaternion components
w	y
x	y
y	-w
z	-w

Table 5.4: Quaternion formed if the aircraft needs to rotate clockwise ($90^\circ < \theta < 180^\circ$)

New Quaternion Components	Consists of previous Quaternion components
w	y
x	y
y	-w
z	-w

Table 5.5: Quaternion formed if the aircraft needs to rotate counter clockwise ($90^\circ < \theta \leq 180^\circ$)

New Quaternion Components	Consists of previous Quaternion components
w	y
x	y
y	-w
z	-w

Table 5.6: Quaternion formed if the aircraft needs to rotate counter clockwise ($0^\circ \leq \theta \leq 90^\circ$)

New Quaternion Components	Consists of previous Quaternion components
w	-y
x	-y
y	w
z	w

Referring to the above tables, it can be noted that Table 5.3, Table 5.4 and Table 5.5 all had the same conversion, while Table 5.6's conversion was different.

- This newly formed quaternion (referred to as the new ArUco quaternion) needed to be used by the aircraft. However, the aircraft currently had an orientation at the time of converting the ArUco marker's quaternion. Therefore, to use this new ArUco quaternion, it needed to be multiplied to the aircraft's current quaternion.

In order to perform quaternion multiplication, it must be noted that quaternion multiplication is not commutative, meaning:

$$a \times b \neq b \times a$$

Therefore, to perform the correct multiplication between the aircraft and the new ArUco quaternion, the ordering was as follows:

$final\ quaternion = aircraft's\ quaternion \times new\ ArUco\ quaternion$

Figure 5.1 shows a screenshot of code obtained from website Euclidean Space, which was used to implement the quaternion multiplication. The x, y, z and w would form the final quaternion, where q1 would be the aircraft's quaternion and q2 would be the new ArUco quaternion.

```
public final void mul(Quat4d q1,Quat4d q2) {
    x = q1.x * q2.w + q1.y * q2.z - q1.z * q2.y + q1.w * q2.x;
    y = -q1.x * q2.z + q1.y * q2.w + q1.z * q2.x + q1.w * q2.y;
    z = q1.x * q2.y - q1.y * q2.x + q1.z * q2.w + q1.w * q2.z;
    w = -q1.x * q2.x - q1.y * q2.y - q1.z * q2.z + q1.w * q2.w;
}
```

Figure 5.1: Code used to implement the quaternion multiplication (Euclidean Space, 2017)

This final quaternion would be published to the ROS topic aircraftpose_pub, where it would eventually be uploaded to the aircraft by using the flyaircraft.cpp node. This final quaternion is what would be used to align the orientation of the aircraft with the ArUco marker's orientation correctly.

The important methods to take note of in the determinepose.cpp script are:

- MoveAircraft
- ConvertArucoQuaternion
- QuaternionMultiply
- RotateAircraft
- RepositionAircraft

There were many other tasks that were implemented in the determinepose.cpp script. Many of these tasks included the human detection system, which is discussed in section 5.2.2.1. One of them was receiving the horizontal and descent velocity from talker_velocity.cpp and talker_descent.cpp and publishing them to the aircraft by making use of MAVROS' ParamSet and ParamPush. ParamSet and ParamPush are features that allow for practically any parameter on the Pixhawk to be adjusted from a script.

It must be noted that, when the aircraft was descending towards the markers and had reached one meter above the marker, the developed system caused the aircraft to switch from Offboard flight mode to Land flight mode, where it would descend directly without adjusting its horizontal position. This was decided upon for two reasons:

1. The marker would become too big to be detected by the vision system, resulting in the ArUco marker not being completely in the video feed.
2. It would force the aircraft to disarm itself once landed.

5.1.3.2 flyaircraft.cpp

As discussed, the flyaircraft.cpp (which can be referred to in Appendix 5.2) node's main purpose was to communicate to the aircraft via MAVROS. Some of these commands consisted of:

- a. Allowing the aircraft to arm and take off.
- b. Telling the aircraft where to re-position itself by sending positional and orientational information.
- c. Instructing the aircraft to change its flight mode.

The node would receive information with regard to position and orientation from the determinepose.cpp node and communicate this to the aircraft. The topics that are subscribed to within the node can be referred to in Table 5.7.

Table 5.7: The topics subscribed to in the flyaircraft.cpp node

Topic	Description of the topic
FlightmodeFromUser	Flight mode received from talker_flightmode.cpp script.
mavros/state	Receives the current state from the aircraft via MAVROS (e.g. whether the aircraft is armed or not).
mavros/global_position/rel_alt	Receives the aircraft's current relative altitude, via MAVROS, from where the aircraft was armed.
determinedaircraftpose	Receives the determine positional and orientational information calculated in determinepose.cpp node.

Table 5.8 displays the published topics that were contained in the flyaircraft.cpp node to be used by the landing system.

Table 5.8: The topics published to in the flyaircraft.cpp node

Topic	Description of the topic
local_pos_pub	Publishes the determined position and orientation to the aircraft via MAVROS.

5.1.3.3 talker_autoland.cpp

This node asks the user to choose how the aircraft should be flown while in offboard mode. The choices were:

- manual: the aircraft would be able to move to a set point based on the co-ordinate and co-ordinate value it received in the talker_coordinates.cpp node.
- hover: the aircraft will reposition itself directly above the ArUco marker and hover (as in maintain its altitude).
- land: the aircraft will perform its autonomous landing sequence.
- rotate: the aircraft will rotate based on a quaternion it has received via the talker_quat.cpp node.
- follow: the aircraft will rotate and reposition itself based on the position of the person detected (this is used for the human detection system).

This node would publish one topic called AutoLandFromUser, where it would contain one of the above offboard flight modes chosen by the user. This node can be referred to in Appendix 5.3.

5.1.3.4 talker_coordinates.cpp

This node, which can be found in Appendix 5.4, was used to control the aircraft manually while the aircraft was in offboard flight mode. The node would publish one topic called CoordinatesFromUser, which contained the axis the aircraft needed to travel along and the co-ordinate to which it should fly, based on the input from the user.

Some examples are:

- z 3
- x -2
- y 5

Only one axis and one coordinate can be published at a time.

[5.1.3.5 talker_descent.cpp](#)

The node, which can be referred to in Appendix 5.5, was used to allow the user to control the descending velocity of the aircraft. This node would publish one topic called `DescentFromUser`, where it would send the velocity (in metres per second) decided upon by the user (e.g. 1.5). The purpose of the node was to be able to control the descending velocity of the aircraft, depending on the speed of the wind on the day of testing the landing system.

[5.1.3.6 talker_velocity.cpp](#)

This node was used to allow the user to choose the horizontal velocity of the aircraft. The node would publish one topic called `VelocityFromUser`, where it would send the velocity (in metres per second) chosen by the user (e.g. 5.0). This node was also developed to be able to control the horizontal velocity of the aircraft, depending on the wind conditions on the day of testing the landing system. The node can be seen in Appendix 5.6.

[5.1.3.7 talker_flightmode.cpp](#)

This node, which can be referred to Appendix 5.7, was used to choose the flight mode for the Pixhawk flight controller. The flight modes that the node allowed were:

- Offboard
- Stabilized

The node would publish to one topic called `FlightModeFromUser`, where it would contain one of the two flight modes above. The landing system would not run until the flight mode of the aircraft had been switch into Offboard mode.

5.1.3.8 talker_quat.cpp

This node was used to rotate the aircraft by a quaternion when the manual mode was selected in talker_autoland.cpp node. The node allowed a user to enter the quaternion in the form (w x y z). The node would publish one topic called QuatFromUser, were it would publish the quaternion that was inserted. This node can be found in Appendix 5.8.

5.1.3.9 listener_status.cpp

This node, which can be seen in Appendix 5.9, was used to listen to the current commands being performed or status of the aircraft throughout the entire system. The node had one subscriber called StatusFromAircraft, where it would wait to receive any information with regards to the system and then output the information to the node. This node was developed for the purpose of using the node on a ground station computer.

5.1.4 Landing System Flow Chart

A flow chart was developed for the landing system. This can be referred to in Appendix 5.15.

5.2 Human Detection System

This section will provide a basic overview of the human detection system as well as the design of the system in detail, discussing the scripts that were developed for this system. A flow chart of the system is also available.

Before getting started, it must be pointed out that the roll, pitch and yaw angles of an aircraft are not the same as the Euler angles of the orientation of the aircraft. Euler angles represent the rotation of an object based on a fixed coordinate frame while the roll, pitch and yaw angles represent the rotation of an object based on the current coordinate frame. However, for the purposes of this section, the aircraft will only be rotated about its z axis (namely, only one axis). This means, with respect to the Euler angle, that the rotation about the z axis can be used to yaw the aircraft by the same angle.

For the rest of section 5.2, with regards to the Euler angle, the rotation about the z axis will be denoted by ψ .

5.2.1 Basic overview of the human detection system

A basic guideline as to how the human detection system works follows:

1. The aircraft's flight mode is switched to Offboard mode via a node.
2. The aircraft will arm itself, take off and become airborne.
3. The aircraft is sent to an altitude of ten metres.
4. A node is run where a command to follow the aircraft is sent, informing ROS to implement the human detection system.
5. A human is detected, providing the bounding box around the person.
6. A node is run to calculate the ground distance between the aircraft and the person as well as the angle required to rotate the aircraft so that the person is in the centre of the screen.
7. A timer will start, where the aircraft will have five seconds to rotate by using the angle determined.
8. Once the timer has finished, coordinates for the aircraft will be determined based on the previously determined ground distance.
9. The aircraft will fly towards the coordinates.

The above steps explain how the aircraft will be able to follow a person.

5.2.2 Design

This system made use of the same nine nodes that were developed for the landing system. However, extra code was added to the node `determinepose.cpp` to allow the aircraft to follow a person. This modification of code will be discussed in section 5.2.2.1.

There were four new nodes that were developed to be used by this system (where two were written in C++ language and two were written in Python language), namely:

1. `boxinfo.cpp`: This C++ script was used to read the bounding box information and determine the ground distance and angle to rotate the aircraft.

2. boundingboxmove.cpp: This C++ script was used to develop coordinates for the aircraft to fly towards, based on the ground distance and the angle of the aircraft.
3. EulerToQuat.py: This Python script was used to convert Euler angles to a quaternion
4. QuatToEuler.py: This Python script was used to convert a quaternion to Euler angles.

The reason for developing the two python scripts was due to the libraries available in Python being better at converting between Euler angles and quaternions, as opposed to C++.

The following design attributes must be noted:

- This system was designed so that the aircraft would maintain its altitude while following a person.
- This system was designed so that only the bounding box information regarding a person was able to be used. This means that only a person would be followed and not, for example, a dog. However, the video feed will still show a bounding box around all detected objects.

5.2.2.1 determinepose.cpp

As previously discussed, this script was also used in the human detection system. There was another topic that was subscribed to in the script. This topic, which can be seen in Table 5.9, will receive the x and y coordinates that the aircraft will need to travel towards.

Table 5.9: The extra topic subscribed to in the determinepose.cpp node

Topic	Description of the topic
MultipleCoordinates	x and y coordinates received from boundingboxmove.cpp.

The extra method that was developed in the script, which was to be used for the human detection system, was called FollowPerson. This method, which can be referred to in

Appendix 5.1, was responsible for arranging when the aircraft needed to rotate to centre the person in the video frame and when the aircraft should travel towards the coordinates. It made use of a timer, where the aircraft had five seconds to rotate during this time. It also made use of the angle determined in the node `boxinfo.cpp` to determine by how much and in what direction the aircraft needed to rotate. After the time had elapsed, co-ordinates were received from the `boundingboxmove.cpp` node and then sent to the aircraft by making use of the `flyaircraft.cpp` node.

5.2.2.2 `boxinfo.cpp`

This node was used for implementing the calculations and formulas determined in sections 4.4.2 and 4.4.3. It was responsible for using the `darknet_ros` package to collect the bounding box information, determine the angle to rotate the aircraft by, as well as determine the ground distance. The node, which can be found in Appendix 5.10, would first determine whether the detected object was a person or not. If it were a person, the angle and ground distance would be calculated by using the bounding box information.

The topic that this node was subscribed can be seen in Table 5.10. Table 5.11 lists all the published topics in the node.

Table 5.10: The topic subscribed to in the `boxinfo.cpp` node

Topic	Description of the topic
<code>darknet_ros/bounding_boxes</code>	The bounding box information of all the detected objects in the video frame by using <code>darknet_ros</code> .

Table 5.11: The topics published to in the `boxinfo.cpp` node

Topic	Description of the topic
<code>AngleFromBoundingBox</code>	The calculated angle and direction that the aircraft must rotate towards in order to centre the person in the middle of the video frame.
<code>DistanceFromBoundingBox</code>	The calculated ground distance that the aircraft is away from the detected person.

5.2.2.3 boundingboxmove.cpp

This node, which can be referred to in Appendix 5.11, was designed to implement the calculations determined in section 4.4.4. The node would calculate coordinates for the aircraft to fly towards based on:

1. The direction the aircraft was facing.
2. The aircraft's current position.
3. The aircraft's home position.
4. The ground distance that was previously calculated.

The aircraft would obtain the direction of the aircraft from the QuatToEuler.py node.

The topics that the node was subscribed to can be seen in Table 5.12.

Table 5.12: The topics subscribed to in the boundingboxmove.cpp node

Topic	Description of the topic
DistanceFromBoundingBox	This is the ground distance that was calculated in boxinfo.cpp.
AngleForLinearMovement	The angle that the aircraft is currently facing obtained from QuatToEuler.py.

The node only had one publisher topic, which can be referred to in Table 5.13.

Table 5.13: The topic published to in the boundingboxmove.cpp node

Topic	Description of the topic
MultipleCoordinates	This will publish the x and y coordinate that the aircraft needs to fly towards.

5.2.2.4 EulerToQuat.py

This node, which can be found in Appendix 5.12, was responsible for converting the angle (as in the angle determined to rotate the aircraft to centre the person in the video feed) to a quaternion for the aircraft to use. A Python library called pyquaternion was used to help with this conversion. The library was able to take Euler angles and convert them into a quaternion. The ψ value (rotation about the z axis) of the Euler angle was the angle obtained and the rotations about the x axis and y axis were both

set to zero as the aircraft was only required to yaw. These Euler angles were then converted to a quaternion and then sent to the main determinepose.cpp node. The topics subscribed and published to can be referred to in Table 5.14 & Table 5.15.

Table 5.14: The topic subscribed to in the EulerToQuat.py node

Topic	Description of the topic
AngleFromBoundingBox	The angle and direction that the aircraft must rotate towards, which is obtained from boxinfo.cpp node.

Table 5.15: The topic published to in the EulerToQuat.py node

Topic	Description of the topic
QuatFromUser	The converted quaternion which was used in the determinepose.cpp node.

5.2.2.5 QuatToEuler.py

This Python node, which can be seen in Appendix 5.13, was responsible for converting quaternions into Euler angles. The library that was used to help with this conversion was the euler_from_quaternion library. The quaternion that was received was the quaternion that was formed by the aircraft's local pose. Only the ψ rotation was extracted from the Euler angle, which was eventually sent to the boundingboxmove.cpp node to be used.

The topics subscribed and published to can be seen in Table 5.16 and Table 5.17.

Table 5.16: The topic subscribed to in the QuatToEuler.py node

Topic	Description of the topic
mavros/local_position/pose	The position and orientation of the aircraft obtained via MAVROS.

Table 5.17: The topic published to in the QuatToEuler.py node

Topic	Description of the topic
AngleForLinearMovement	The aircraft's current ψ rotation, which was used by the boundingboxmove.cpp node.

5.2.3 Human detection system flow chart

A flow chart was developed for the human detection system. This can be referred to in Appendix 5.16.

5.3 User Requirements Specification

The user requirements specification (URS) for this system are:

- The aircraft will be able to fly manually and autonomously.
- The vision system will detect an ArUco marker.
- The vision system will detect a human and be able to distinguish it from other objects.
- The aircraft and the vision system will be integrated with Robot Operating System (ROS).
- The aircraft may land on an ArUco marker autonomously.
- Using the developed landing system may result in the aircraft landing more accurately compared to when the aircraft makes use of the stand GPS based landing system.
- Using the human detection system may result in the aircraft's position and orientation being manipulated, allowing the detected human to be centred in the middle of the video feed.

5.4 Conclusion

In this chapter, the entire system was discussed, based on how ROS was used to integrate the system. The nodes and published and subscribed topics for both the landing system and the human detection system were discussed in detail. Flow charts for both the landing system and the human detection system were presented. The next chapter, Simulation, will discuss how the landing system and the human detection system were simulated using Gazebo.

Chapter 6: Simulation

The purpose of this chapter is to discuss the simulation of the landing system and the human detection system that was performed. The setup of the simulation will be discussed and the benefits to using such simulation.

6.1 Setup of the Simulation

To help with the testing phase of the system, it was decided that the system would be tested on a simulator. Research was performed in the type of simulation software available for use. The software that was chosen for this research was Gazebo, which is an open-source 3D simulation environment for autonomous robots. Gazebo is useful for testing object-avoidance and computer vision systems (PX4, 2019g). Since Gazebo is open-source, it has been integrated into ROS' main installation by developers.

There are multiple benefits to testing the system in Gazebo first before performing real life testing of the system. Some of these benefits are:

- A system can quickly be developed and tested on the simulation.
- A simulation allows for code developed to be tested thoroughly before testing it on the simulation. This will heavily reduce the number of crashes that could occur on the actual system, thus reducing the cost of having to purchase replacement parts.
- Parts of a system can be individually tested in Gazebo (e.g. the camera system on detecting ArUco markers can be tested).
- It allows systems that incorporate Artificial Intelligence (AI) to train their neural networks on the simulation.

To use the model with ROS, a Software in the Loop (SITL) simulation environment had to be set up. A model would need to be created for use in the simulation software. A model is an object that is manipulated in the simulation. In the case of this research, it would be an aircraft. Developers from the PX4 team had developed a SITL simulation environment to be used for testing. The developers had made multiple models to be used in Gazebo. One of models that they had developed was a quadcopter, referred to in Figure 6.1, which makes use of PX4's firmware.

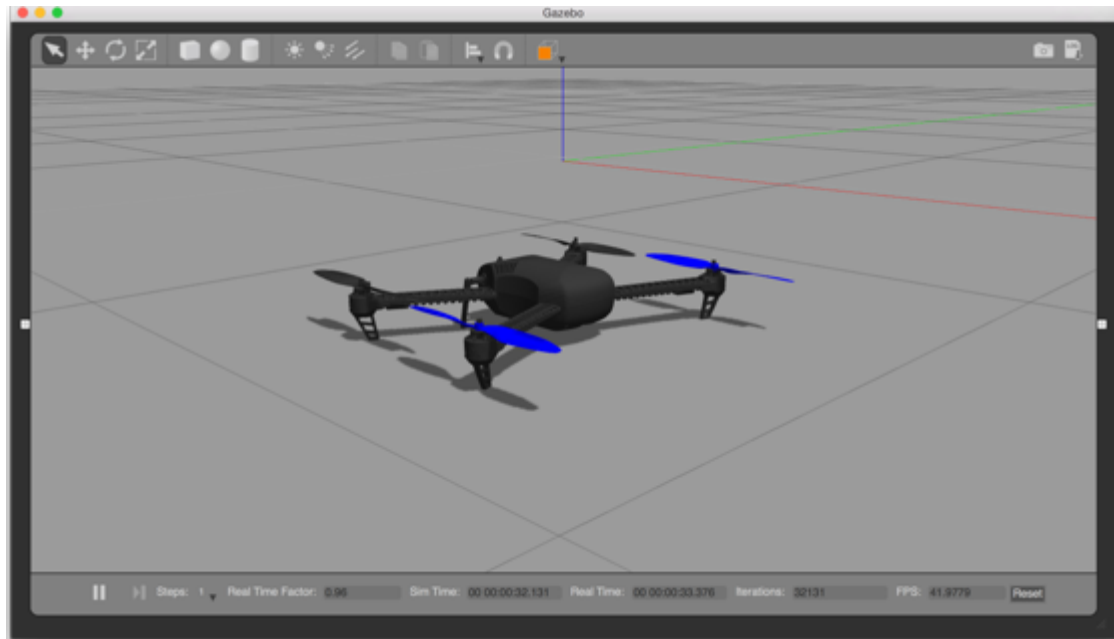


Figure 6.1: Quadcopter model developed by PX4 for Gazebo

One aspect that was missing from this model in Gazebo was a camera. A guide was developed by Aerial Robotics, where they demonstrate how to adapt the model to add a cube underneath the fuselage (Aerial Robotics, 2018). This cube was configured to act as a camera.

It must be noted that the simulations were performed on a desktop PC that contained an Intel i7 processor and an AMD Radeon graphics card. This was done because the desktop PC was able to provide more processing power than the two laptops that were used throughout this research. However, it came with a disadvantage because the `darknet_ros` package was designed to work on a Nvidia graphics card. This meant that all the human detection video processing was performed on the computer's processor and not on the graphics card. The result was that the frames per second (FPS) were constantly under 1 FPS.

Another point to note is that the simulation was only used to observe how the aircraft would react to the developed algorithms. No actual data would be collected from the simulation.

6.2 Landing System Simulation

To test the landing system in Gazebo, an ArUco marker model needed to be added to the Gazebo environment (referred to as the world). A few ArUco markers for Gazebo had previously been developed by Jose Luis Sanchez Lopez, which he had available for use on his GitHub page (Lopez, 2016). Aerial Robotics provided a guide on how to add one of his developed markers into Gazebo (Aerial Robotics, 2018).

For the simulation, an ArUco marker with an ID of 7 was used, which had dimensions of 500 x 500 mm. An example of what the setup resembled can be found in Figure 6.2.

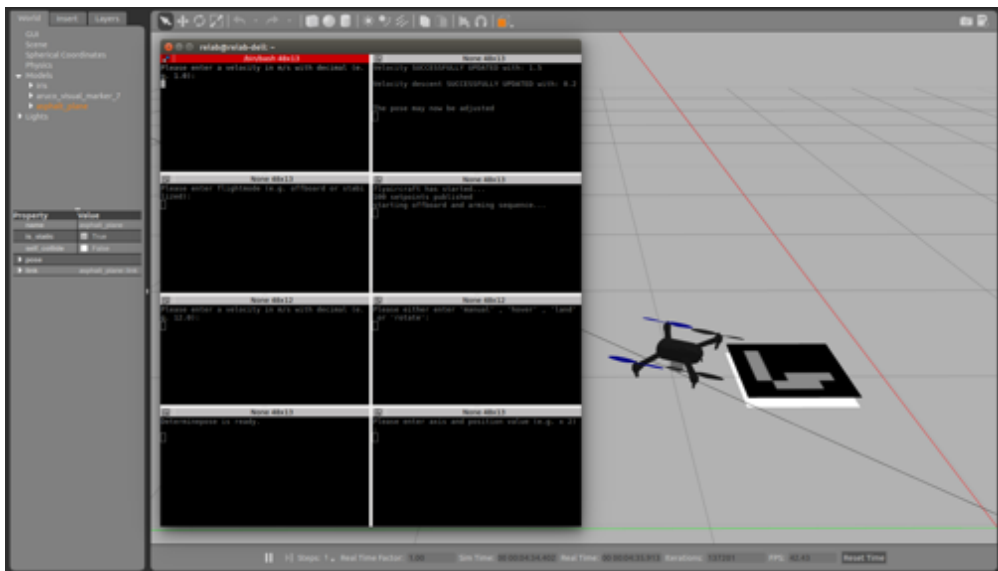


Figure 6.2: The Gazebo setup for the landing system.

To perform the simulation, the packages to be used (e.g. aruco_ros, MAVROS, etc.) and the package containing all the self-developed nodes were required to be run. The terminals that were used to run each of the respective packages or nodes for the landing system can be found in Appendix 6.1. The terminals were all grouped by making use of the Terminator application for Linux (as discussed in section 3.3.4.3). To allow the aircraft to take off, the word “offboard” was entered into the talker_flightmode.cpp node, which switched the aircraft to offboard mode. The aircraft armed automatically and the aircraft became airborne. In a separate window, the video feed of the camera could be seen. Figure 6.3 resembles the aircraft that has taken off and the window on the right shows the video feed.

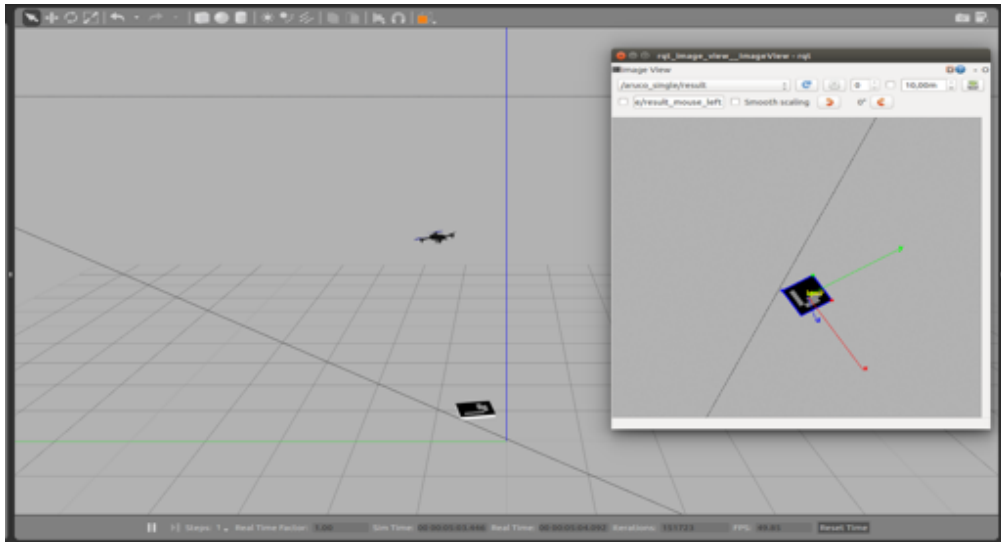


Figure 6.3: The aircraft has taken off, where the window on the right shows the camera feed.

In order to use the landing system, the vision system was required to be able to detect the marker clearly, meaning the whole square shape of the marker needed to be present in the video feed. If a marker was detected, the marker's x, y and z axes became present. The colour coding of the axes were as follows:

- X axis: red
- Y axis: green
- Z axis: blue

In Figure 6.3, the window shows the ArUco marker that was detected, where the ArUco marker was positioned on the ground. In order for the aircraft to rotate so that it aligned itself correctly with the ArUco marker, the aircraft would need to rotate clockwise by an angle smaller than 90° so that the y axis (green arrow) in the window pointed vertically upwards. Figure 6.4 shows the camera feed of when the aircraft had rotated to align itself with the marker correctly. As discussed in section 5.1.3.1, there are four scenarios in which the aircraft would rotate, namely:

1. Rotate clockwise by an angle smaller than or equal to 90° .
2. Rotate clockwise by an angle greater than 90° and smaller than 180° .
3. Rotate counter-clockwise by an angle smaller than or equal to 90° .
4. Rotate counter-clockwise by an angle greater than 90° and smaller than or equal to 180° .

Figure 6.5 shows a window of the camera feed for cases 1 and 2, while Figure 6.6 shows the camera feed for cases 3 and 4.

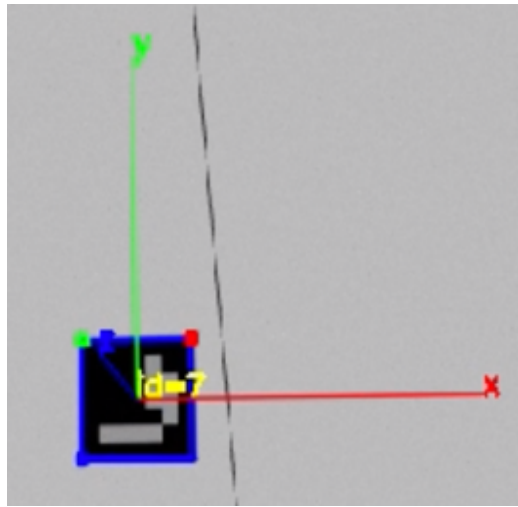


Figure 6.4: Camera feed on when the aircraft had correctly aligned itself with the ArUco marker

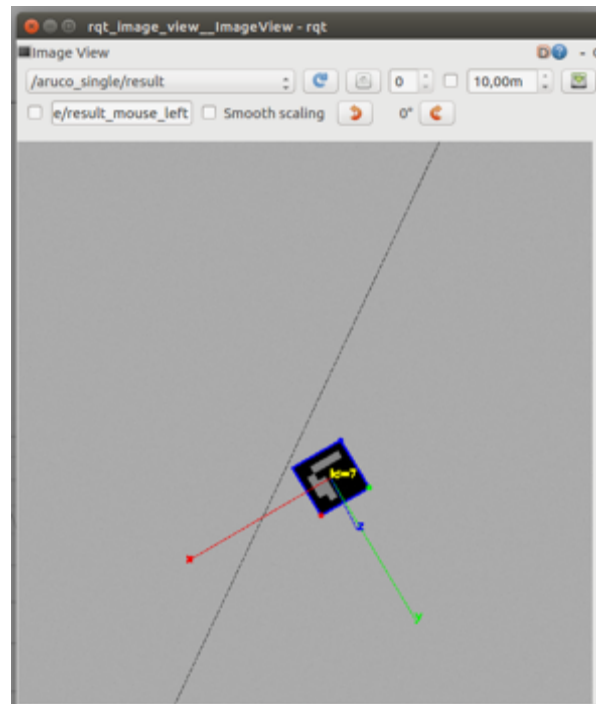
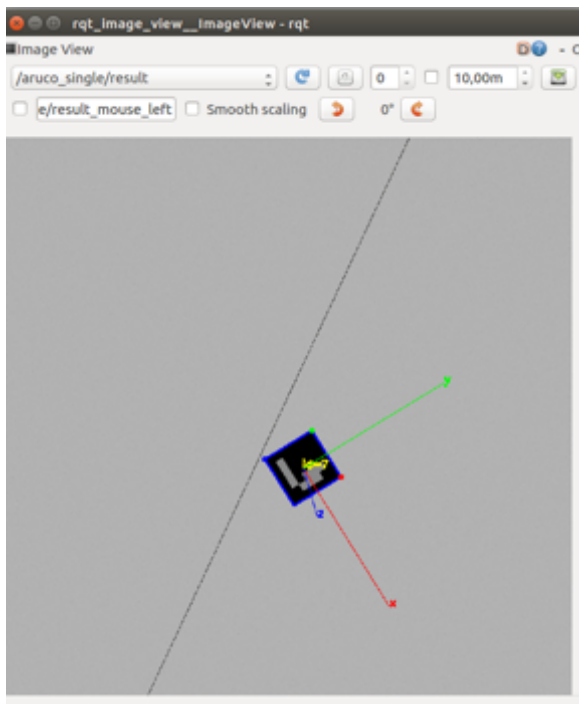


Figure 6.5: (a) Case 1 (b) Case 2

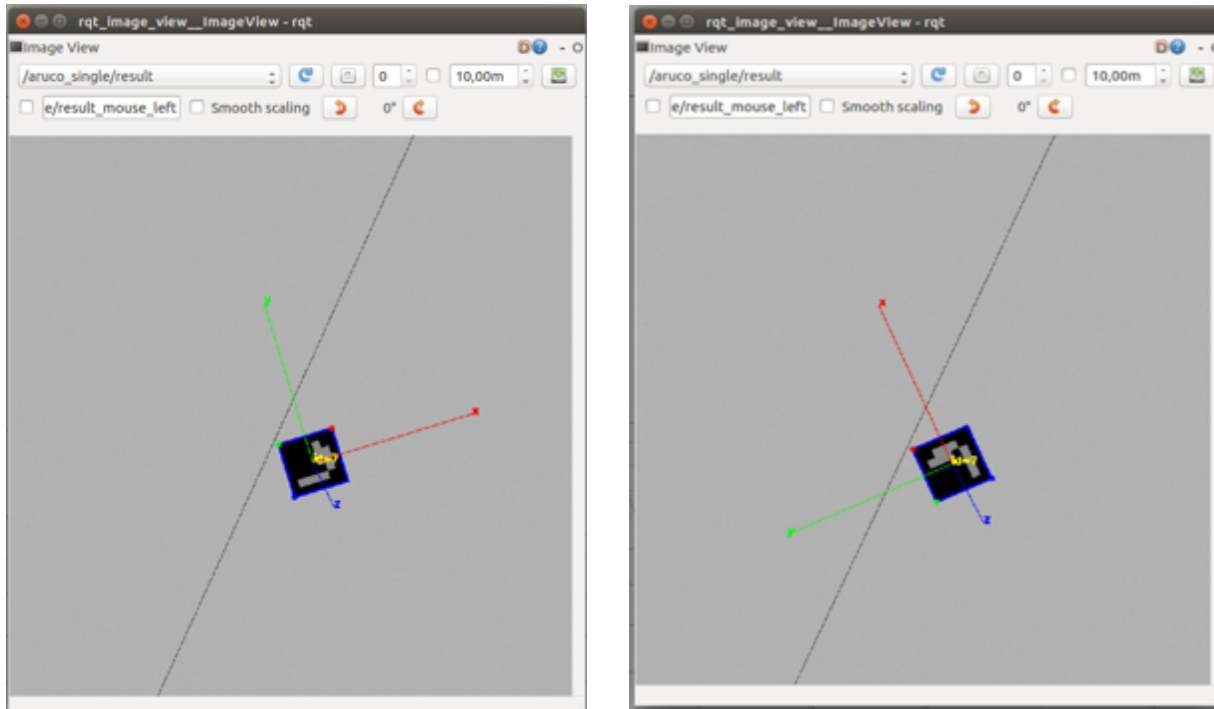


Figure 6.6: (a) Case 3 (b) Case 4

For illustration purposes, case 2 will be used. Once the aircraft was airborne and had detected the ArUco marker, the system was allowed to land. This was done by inserting the word, “land” into the `talker_autoland.cpp` node. An example of this landing process for case 2 can be found in Appendix 6.3.

A node tree was developed for all the nodes that were used to test the landing system in Gazebo. The nodes tree, which can be referred to in Appendix 6.4, shows the communication between all the nodes used. Video footage of the simulation was recorded for all four cases. The videos for each case, which were uploaded to YouTube, can be found at :

- Case 1: <https://youtu.be/mdYK1TQIlvM>
- Case 2: <https://youtu.be/ES7XcSzqHwE>
- Case 3: <https://youtu.be/frJMAQSdxZw>
- Case 4: <https://youtu.be/HRgFgTyTsel>

These videos can only be accessed via the links as the videos have been set to be unlisted to the general public.

6.3 Human Detection System Simulation

To perform a simulation for the human detection system, the same setup was used that was developed for the landing system. However, there were an additional six nodes that were executed. These additional nodes can be found in Appendix 6.2. The simulation consisted of running a pre-recorded video of a person walking on a field. This was done by using the `video_stream_opencv` package, where it would be given the file directory of a video or package and output the footage of picture to a specific topic. The video that was recorded for the simulation was done by using a DJI Phantom 4 Advanced in 1920x1080 resolution (Full HD), where the camera angle of the aircraft was set at 45°. The video (as in the published topic) can be referred to in Figure 6.7(a). This topic was then used by the `darknet_ros` package. This means that for the simulation, the bounding box, that was formed using the `darknet_ros` package, will fall within the follow criteria:

- The Xmax value will not be bigger than 1920.
- The Xmin value will not be smaller than 1.
- The Ymax value will not be bigger than 1080.
- The Ymin value will not be smaller than 1.

Figure 6.7(b) shows the bounding box that was determined for the video that was published.



Figure 6.7: (a) Video of a person walking that was published by the `video_stream_opencv` package (b) bounding box that was formed using the `darknet_ros` package

It must be noted that both pictures were captured at the exact time. However, the position of the person in both figures is not the same. This is due to the low FPS that was discussed in section 6.1. The final setup for the human detection system can be found in Figure 6.8.

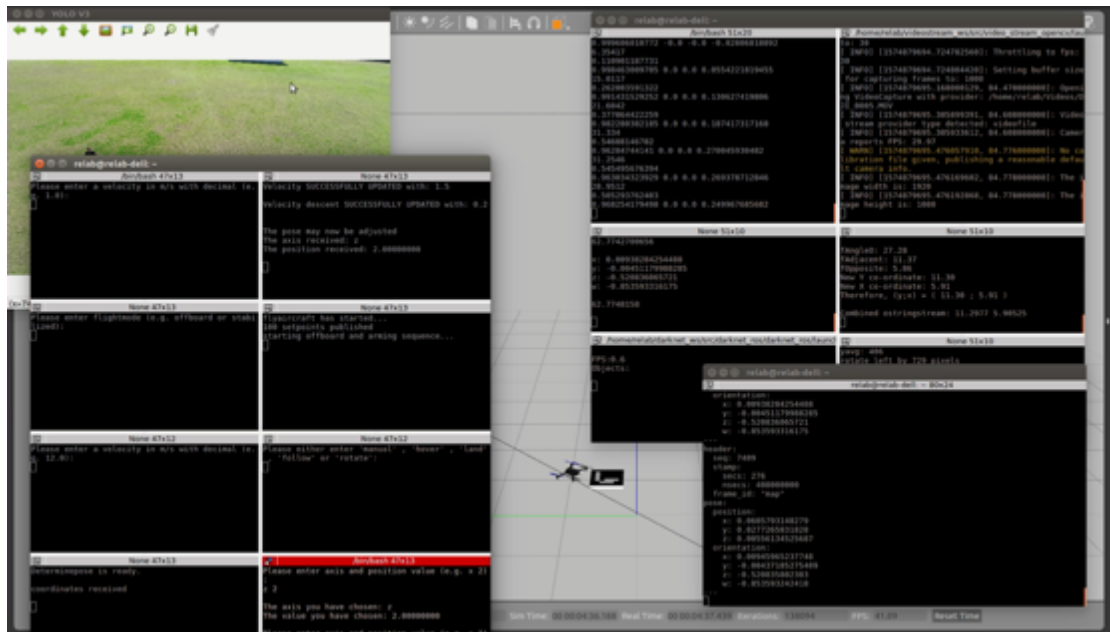


Figure 6.8: The Gazebo setup for the human detection system

For this simulation, the aircraft was first switched to Offboard mode, where it would arm automatically and then take off to an altitude of two metres (designed for the landing system). However, the human detection system was designed to work at an altitude of ten metres. This meant that the aircraft was first manually adjusted to an altitude of ten metres by using the `talker_coordinates.cpp` node. Once the aircraft was in position, the aircraft was told to follow the person via the `talker_autoland.cpp` node. The human detection system would then execute, allowing the aircraft to rotate, based on the angle that was determined, as well as allow the aircraft to reposition itself according to the desired calculated co-ordinates.

There were instances in which the aircraft would rotate:

1. The bounding box that was determined is to the left of the midpoint of the screen.
2. The bounding box that was determined is to the right of the midpoint of the screen.

Case 1 would result in the aircraft yawing in a counter clockwise direction, whereafter it would reposition itself according to the calculated co-ordinates. Case 2 would result in the aircraft yawing in a clockwise direction, whereafter it would reposition itself according to the calculated co-ordinates. Refer to Figure 6.9 and Figure 6.10 to see an illustration of case 1 and Figure 6.11 and Figure 6.12 to see an illustration of case 2. The blue propellers of the aircraft are at the nose of the aircraft (as in the front of the aircraft).



Figure 6.9: Case 1, where (a) is the orientation of the aircraft before the rotation and (b) is the orientation of the aircraft after the rotation

```
pose:
  position:
    x: 0.0257942546159
    y: -0.0339961014688
    z: 10.0336790085
  orientation:
    x: 0.0117031393312
    y: -0.000143611435034
    z: -0.516043842605
    w: -0.856482271906
---
```

```
pose:
  position:
    x: 5.82904815674
    y: 10.923620224
    z: 9.91774177551
  orientation:
    x: 0.0165362119632
    y: 0.00145704421189
    z: -0.703106790406
    w: -0.710890558634
---
```

Figure 6.10: Case 1, where (a) is the pose of the aircraft before the repositioning and (b) is the pose of the aircraft after the repositioning



Figure 6.11: Case 2, where (a) is the orientation of the aircraft before the rotation and (b) is the orientation of the aircraft after the rotation

```
pose:
  position:
    x: 5.75479984283
    y: 10.8742656708
    z: 10.0100669861
  orientation:
    x: 0.0164825392687
    y: -0.00232137070773
    z: -0.702993052855
    w: -0.711002020256
---
```

```
pose:
  position:
    x: 5.74864673615
    y: 23.0196113586
    z: 10.0840339661
  orientation:
    x: 0.0160357243933
    y: -0.0103847967757
    z: -0.579404359009
    w: -0.814816255717
---
```

Figure 6.12: Case 2, where (a) is the pose of the aircraft before the repositioning and (b) is the pose of the aircraft after the repositioning

A node tree was also developed for all the nodes that were used to test the human detection system in Gazebo. This nodes tree can be referred to in Appendix 6.5. Video footage of the simulation was recorded, where case 1 and case 2 were both executed. The video shows the aircraft changing its orientation as well as its position, first for case 1 and then for case 2. This video can be found at: <https://youtu.be/5Sr8a3fw3kc>.

6.4 Conclusion

The setup and importance of the simulation were discussed. The method of testing the landing system was explained, where figures were used to show the system operating successfully. The method of testing the human detected system was also discussed, providing figures (including figures of the aircraft's positional and orientational information). The next chapter, Testing and Discussion, will discuss the testing that was performed as well as provide a data analysis on the landing system, comparing the GPS based landing system with the developed ArUco marker landing system.

Chapter 7: Testing and Discussion

This chapter will discuss the numerous types of test flights that were physically performed on the multirotor aircraft. A data analysis will be performed, where the aircraft's current GPS system to land the aircraft will be compared with the developed landing system that makes use of the ArUco marker. The human detection system will be discussed to explain the testing that was performed on the simulation computer. Finally, improvements to the system will be discussed.

It must be noted that there was an issue with the PX4 firmware at the time of performing the test flights. The issue consisted of the aircraft not disarming itself once the aircraft hand landed on the ground. ROS specialists from FH Aachen, who also integrate ROS with multirotor aircrafts, were contacted about the issue. They explained that they also encountered the same problem and that it was most likely a firmware issue. This was reported to PX4. However, no feedback had been received at the time of writing. In the meantime, in order to perform the test flights, the aircraft had to be manually disarmed for all autonomous missions and landing (including the GPS based landing system and the developed ArUco marker landing system). This manual disarming was performed by the pilot of the aircraft, Mr Paul Mooney, who was an advisor for the research. It resulted in the aircraft not landing at the exact point specified by the algorithms, resulting in an inaccuracy (± 150 mm) for the landing system.

All test flights were performed at Port Elizabeth Radio Flyers (PERF). The aircraft flying at the location can be found in Figure 7.1. The videos referred to in this chapter were recorded either on an Apple iPhone 10 smart phone, a DJI Phantom 4 Advanced drone or a DJI Mavic drone.

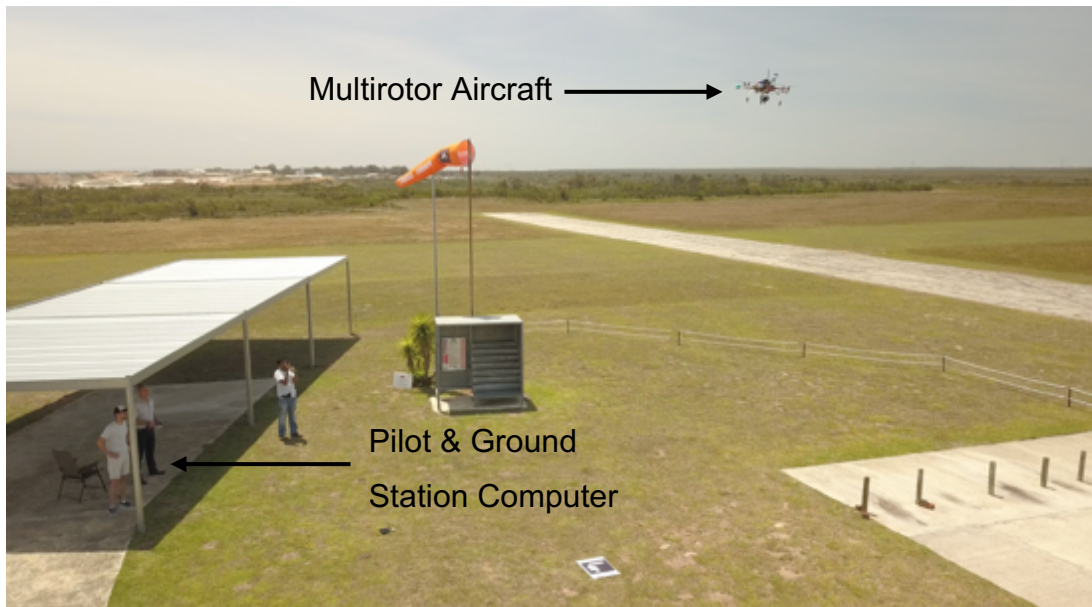


Figure 7.1: The multirotor aircraft flying at PERF

7.1 Test Flights

The test flights that were performed on the aircraft include:

- A manual test flight.
- An autonomous test flight performing a mission.
- An autonomous test flight where a mission is injected into the flight controller by a separate system.
- An autonomous test flight using ROS.
- The aircraft autonomously landing using ROS.

7.1.1 Manual Test Flight

A manual test flight was performed, where the pilot was able to fly the aircraft by using the transmitter to control the aircraft. To perform this test, the aircraft needed to be flown by the pilot. For the test flight, the aircraft was flown in three flight modes, which were:

1. Acro
2. Stabilized
3. Position

For the test, the aircraft took off in Stabilized flight mode, where it was able to level itself unassisted. After a few seconds, the aircraft was switched to Acro flight mode,

which resulted in the pilot having to level the aircraft instead. Finally the aircraft was switched to Position flight mode, where the aircraft was able to hold its position and altitude by using the GPS module that was mounted onto the aircraft. The aircraft was eventually landed by using the same flight mode.

A video of the manual flight test was recorded on an iPhone. The video consists of the pilot explaining how he is controlling the aircraft. This video can be found at: <https://youtu.be/8vMhNxAXXFs>.

7.1.2 Autonomous test flight performing a mission

A mission was performed by the aircraft, where the aircraft would take off, fly to a set of GPS coordinates specified on the ground station computer and land back at the point where the aircraft took off. To perform this mission, the aircraft made use of the GPS module to direct the aircraft to the required GPS coordinates. The mission that was setup on QGroundControl for the aircraft to perform can be found in Figure 7.2.



Figure 7.2: The mission setup for PERF that was done on QGroundControl

The aircraft would take off at point 1, fly towards point 2, then point 3 and so on until reaching point 6. Point 7 was a command to return home along the quickest route possible and land at the take-off position. The aircraft managed to perform the autonomous mission successfully. The aircraft's flight route that was performed could

be observed on QGroundControl, which is represented by a red line. A screenshot of the above can be found in Figure 7.3. From the Figure 7.3, it can be seen that the aircraft was slightly off the orange line between points 3 and 4 and points 5 and 6. Point 9 can be ignored as it was a land command inserted for safety reasons.

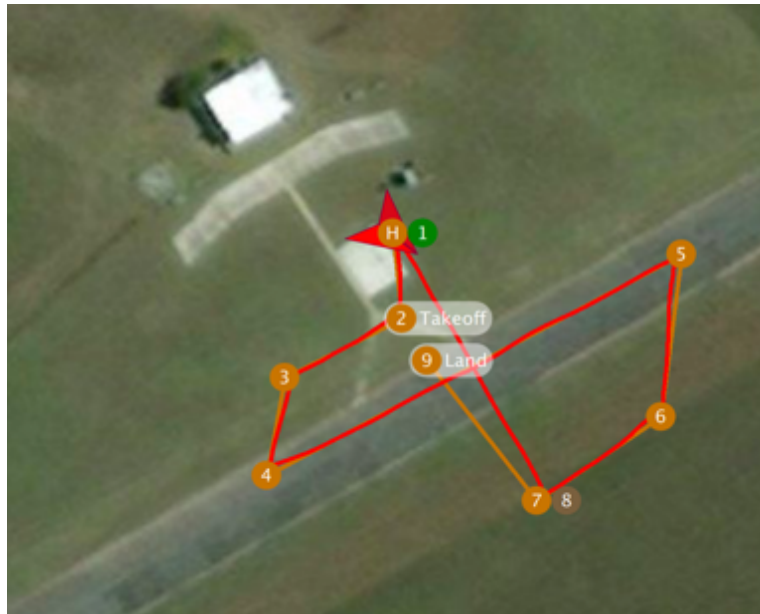


Figure 7.3: The mission performed that was seen on QGroundControl.

A video of the aircraft performing the mission was recorded by using the DJI Mavic drone, where it followed the flight executed by the multirotor aircraft. This video can be found at: https://youtu.be/0u_R5Ex_9bA.

7.1.3 An autonomous test flight where a mission is injected into the flight controller by a separate system

During the early stages of this research, it was suggested by the advisor Mr Paul Mooney that the possibility be investigated to develop a system or alternative method that could control the aircraft or execute a mission. Research was performed in potentially developing a system that would make use of an Arduino.

A system was eventually developed in collaboration with Mr James Sewell, a fellow Masters student who was also performing research in UAVs. An example script developed by David Hasko, which was obtained off the internet, would send a PPM signal out of a 3.5 mm headphone jack that was attached to an Arduino (Hasko, 2016).

This system consisted of using an Arduino Mega, where the board would send a PPM signal through to the transmitter's auxiliary port (which is the same type of port as a 3.5 mm port). Connected to the Arduino was a 3.5 mm headphone jack that plugged into the transmitter as well as three push buttons that were soldered to a Veroboard. The three buttons represented:

- Button 1 soldered to a green wire: Changing the aircraft's flight mode to Auto flight mode.
- Button 2 soldered to a white wire: Changing the aircraft's flight mode to Acro flight mode.
- Button 3 soldered to a red wire: Changing the aircraft's flight mode to Return To Land (RTL) flight mode.

The code was edited to use the buttons only to adjust the PWM signal of one of the channels. This meant that, if a button were pressed, the PWM signal connected to the respective channel would only change. However, the script allowed for adjustments to be made on multiple channels. Figure 7.4 reveals the setup of the Arduino with the 3.5 mm headphone jack and the three push buttons. It must be noted that this system was tested when the aircraft was still using the ArduPilot firmware. The test performed successfully. However there was no video footage recorded to show the system working. The edited Arduino script can be found in Appendix 7.1.

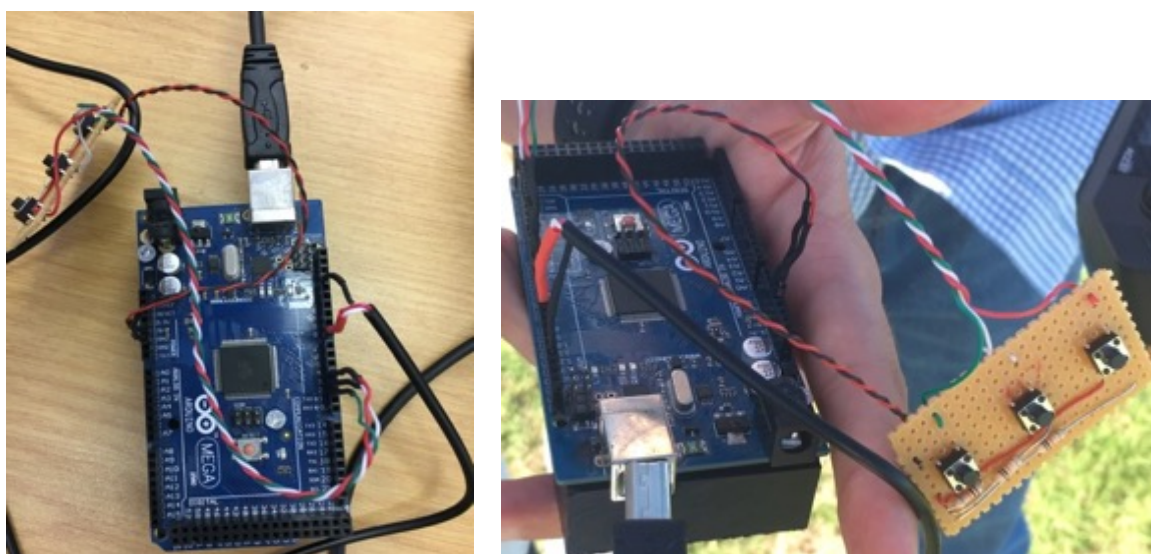
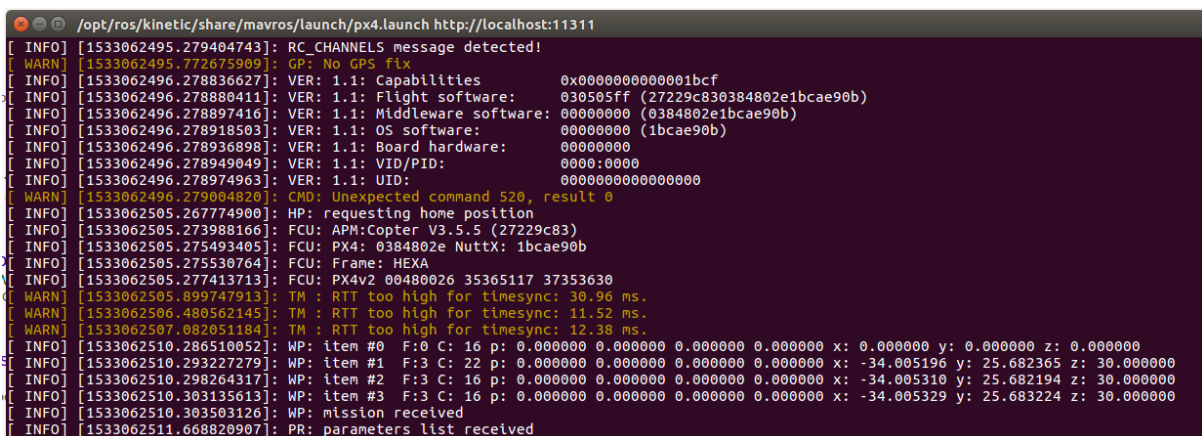


Figure 7.4 The Arduino injection setup

7.1.4 An autonomous test flight using ROS

This test investigated the possibility of injecting new waypoints into the flight controller by using ROS. The setup for this test consisted of developing a ROS script called `InsertWaypoints.cpp`, which consisted of using the MAVROS package. This script can be viewed in Appendix 7.2. The idea behind the script came from a portion of code uploaded to a GitHub forum by a user named `aykutkabaoglu` (GitHub, 2017). For this test, the script consisted of a few GPS co-ordinates that were previously stored. When the script was run, the GPS co-ordinates would be inserted into the flight controller. The GPS coordinates inserted could be seen in the MAVROS terminal on the ground station computer as well as on QGroundControl after performing the mission. The test performed successfully. However, it was never executed on a test flight. The GPS co-ordinates inserted were three co-ordinates at the Nelson Mandela University fields. A screenshot of the MAVROS terminal receiving the new GPS coordinates can be found in Figure 7.5 where the three waypoints can be found at WP: Item #1 – WP: Item #3.



```
/opt/ros/kinetic/share/mavros/launch/px4.launch http://localhost:11311
[ INFO ] [1533062495.279404743]: RC_CHANNELS message detected!
[ WARN ] [1533062495.772675909]: GP: No GPS fix
[ INFO ] [1533062496.278836627]: VER: 1.1: Capabilities: 0x00000000000001bcf
[ INFO ] [1533062496.278880411]: VER: 1.1: Flight software: 030505ff (27229c830384802e1bcae90b)
[ INFO ] [1533062496.278897416]: VER: 1.1: Middleware software: 00000000 (0384802e1bcae90b)
[ INFO ] [1533062496.278918503]: VER: 1.1: OS software: 00000000 (1bcae90b)
[ INFO ] [1533062496.278936898]: VER: 1.1: Board hardware: 00000000
[ INFO ] [1533062496.278949049]: VER: 1.1: VID/PID: 0000:0000
[ INFO ] [1533062496.278974963]: VER: 1.1: UID: 0000000000000000
[ WARN ] [1533062496.279004820]: CMD: Unexpected command 520, result 0
[ INFO ] [1533062505.267774900]: HP: requesting home position
[ INFO ] [1533062505.273988166]: FCU: APM:Copter V3.5.5 (27229c83)
[ INFO ] [1533062505.275493405]: FCU: PX4: 0384802e NuttX: 1bcae90b
[ INFO ] [1533062505.275530764]: FCU: Frame: HEXA
[ INFO ] [1533062505.277413713]: FCU: PX4v2 00480026 35365117 37353630
[ WARN ] [1533062505.899747913]: TM : RTT too high for timesync: 30.96 ms.
[ WARN ] [1533062506.480562145]: TM : RTT too high for timesync: 11.52 ms.
[ WARN ] [1533062507.082051184]: TM : RTT too high for timesync: 12.38 ms.
[ INFO ] [1533062510.286510052]: WP: item #0 F:0 C: 16 p: 0.000000 0.000000 0.000000 0.000000 x: 0.000000 y: 0.000000 z: 0.000000
[ INFO ] [1533062510.293227279]: WP: item #1 F:3 C: 22 p: 0.000000 0.000000 0.000000 0.000000 x: -34.005196 y: 25.682365 z: 30.000000
[ INFO ] [1533062510.298264317]: WP: item #2 F:3 C: 16 p: 0.000000 0.000000 0.000000 0.000000 x: -34.005310 y: 25.682194 z: 30.000000
[ INFO ] [1533062510.303135613]: WP: item #3 F:3 C: 16 p: 0.000000 0.000000 0.000000 0.000000 x: -34.005329 y: 25.683224 z: 30.000000
[ INFO ] [1533062510.303503126]: WP: mission received
[ INFO ] [1533062511.668820907]: PR: parameters list received
```

Figure 7.5: MAVROS terminal showing three new waypoints injected into the flight controller

7.1.5 The aircraft autonomously landing using ROS

This test was performed on the aircraft after successfully testing the landing system on the simulation in Gazebo. For this test, the ArUco marker was printed and fastened to a piece of hardboard using masking tape. Originally, the size of the ArUco marker printed was 500 x 500 mm, the same size as the marker in the simulation. However, after the first test flight, it became apparent that the marker was too big to be landed

on by the aircraft. However, the marker was still able to be detected from above when the aircraft had an altitude of roughly five metres. To help with the landing, the ArUco marker was reprinted to a smaller size, which now had a size of 396 mm x 396 mm. The setup for the test can be found in Figure 7.6.



Figure 7.6: The aircraft and the ArUco marker before take off

To perform the test, the same scripts that were developed for the simulation were used. However, a slight adjustment was made in order to test the system safety. The script `flyaircraft.cpp` was adjusted not to activate Offboard mode automatically when selected in the `talker_flightmode.cpp` node, but rather to wait for an input from the pilot. The Offboard flight mode was setup on the transmitter. This allowed the pilot to exit Offboard flight mode whenever the aircraft was struggling to fly due to the wind or other factors affecting it. For example, at the beginning of testing the Offboard flight mode, the aircraft was instructed to take off to an altitude of five metres. However, since the GPS module had not connected to enough satellites in time, the GPS module was not ready to be used, resulting in the aircraft only using the onboard sensors such as IMU to hover the aircraft at the correct setpoint. This resulted in the aircraft drifting sideways in one direction. So, the pilot had to take over the flying of the aircraft and land it safely to enable the test to be restarted.

To demonstrate that the aircraft was able to detect the marker correctly, a person was told to hold the ArUco marker in his hands and walk a few steps on the grass. The aircraft was changed to “hover” in the `talker_autoland.cpp` node. The aircraft successfully managed to keep hovering above the marker, regardless of where the person was standing. A photo of this can be referred to in Figure 7.7.



Figure 7.7: The multirotor aircraft hovering above the ArUco marker when a person held the marker

After testing the aircraft’s hovering capabilities, it was decided to test whether the aircraft could execute the landing system successfully. For this test, the aircraft was not told to rotate to align itself correctly with the ArUco marker. Instead, the aircraft just repositioned itself above the marker and begin its descent to the ground. Initially, when the script was developed, the aircraft was told to land by setting the z axis of the

coordinates to zero. When it reached the last metre above the ground, the aircraft would stop repositioning itself above the marker and only descend towards the ground.

When the “land” mode was executed in the `talker_autoland.cpp` node, the aircraft managed to reposition itself above the marker and descend towards to the ground successfully. However, the aircraft reached a point roughly 300 mm above the marker, where it would stop descending and remain hovering above the marker (refer to Figure 7.8(a)). After a couple of seconds, the aircraft would disarm, resulting in the brushless motors shutting down and the aircraft dropping out of the air (refer to Figure 7.8(b)). The aircraft managed to disarm because the script contained code to disarm the aircraft when the altitude of the aircraft was zero for a certain period of time. The expected issue was that the aircraft’s sensors predicted the aircraft was already on the ground. To resolve this, the `flyaircraft.cpp` node was adjusted, where the aircraft would switch from Offboard flight mode to Land flight mode once the aircraft had dropped to an altitude of one metre (as shown in the flow chart in Appendix 5.15). This adjustment managed to resolve the problem. However, the aircraft still experienced the disarming issue as specified in the beginning of the chapter.

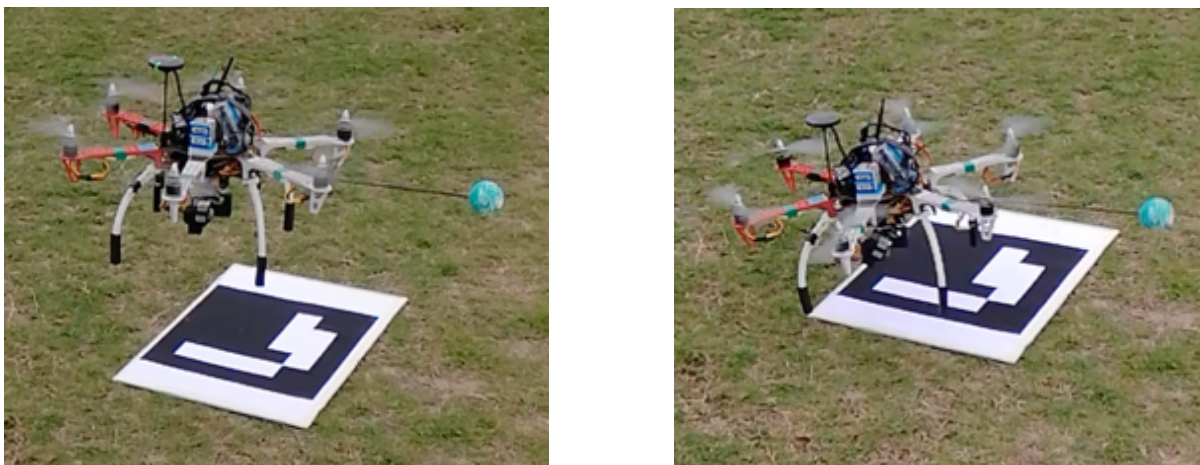


Figure 7.8: (a) aircraft hovering above the marker (b) aircraft disarming, resulting in the aircraft dropping out of the air

After adjusting the code to switch the aircraft to Land flight mode as well as adding in the code to rotate the aircraft correctly, the code was tested on the simulation (as discuss in section 6.3). After successful testing on the simulation, the test was

performed on the aircraft at PERF, where it was successful. Pictures from the test flight can be found in Appendix 7.3.

Videos of the developed landing system and the GPS based landing system being executed were recorded using the DJI Mavic drone. The videos can be found at:

- Developed landing system: <https://youtu.be/7oPOAwui09k>.
- GPS based landing system: <https://youtu.be/WQKi86ONiw4>.

After achieving a successful test result, the decision was made to compare the developed landing system to the aircraft's current method of using the GPS module to land the system. To perform the data analysis, a test was constructed where there would be ten landings performed for each landing method. For each landing, the type of data recorded can be found in Table 7.1. It must be noted that the timer was only started once the aircraft started its descent. This means that the time taken to reposition the aircraft above the point where the aircraft was armed as well as the time taken to rotate the aircraft was not included.

The following steps were implemented for testing each landing method:

1. The aircraft would take off and reach an altitude of five metres, where it would hover.
2. The aircraft would be horizontally repositioned between three and five metres away from the place where the aircraft needed to land. The position chosen was different for each landing.
3. The aircraft was instructed to land.
4. The aircraft repositioned itself above the landing point, where it would orientate itself correctly.
5. The timer was started once the aircraft began to descend.
6. The timer was stopped once the aircraft touched the ground.
7. The accuracy, bearing, time and heading of the aircraft were recorded.
8. This process was implemented ten times to record ten landings.

Table 7.1: Type of data recorded for each landing

Type of Data Recorded	Units	Description
Accuracy	Millimetres (mm)	The distance from where the aircraft took armed to where the aircraft landed.
Bearing	o'clock	The position where the aircraft landed from where the aircraft was armed.
Time	Seconds (s)	The time taken from when the aircraft begins descending to when the aircraft touches the ground.
Heading	Degrees (°)	The direction the aircraft was facing once it had landed.

The initial heading of the aircraft was recorded before commencing the testing for the GPS enabled landing as well as for the developed landing system. These initial headings can be found in Table 7.2.

Table 7.2: The initial heading of the aircraft recorded before testing each landing method

Landing Method	Initial Heading
GPS enabled landing	180°
Developed ArUco marker landing system	179°

Table 7.3 shows the data that was recorded for the aircraft landing via the GPS module and Table 7.4 shows the data that was recorded for aircraft landing via the developed landing system. Figure 7.9 is a plot containing the accuracy for both landing systems and Figure 7.10 is a plot for the time taken to land the aircraft for each landing system.

Table 7.3: Data recorded for the landing of the aircraft by using the GPS module

Landing	Accuracy (mm)	Bearing (o'clock)	Time (s)	Heading (°)
1	990	9	8.9	185
2	4570	10	11.5	175
3	120	9	9.8	182
4	1180	9	9.5	185
5	1510	10	8.6	190
6	700	9	10.5	178
7	1150	4	8.5	184
8	640	3	7.9	173
9	940	3	9.6	176
10	1520	4	10.7	186

Table 7.4: Data recorded for the landing of the aircraft by using the developed landing system

Landing	Accuracy (mm)	Bearing (o'clock)	Time (s)	Heading (°)
1	40	9	34.4	158
2	100	9	18.5	177
3	80	8	20.7	193
4	50	7	20.1	175
5	240	3	23.4	180
6	90	10	24.6	172
7	110	10	20.9	187
8	70	9	22.1	182
9	130	4	20.8	188
10	100	9	23	180

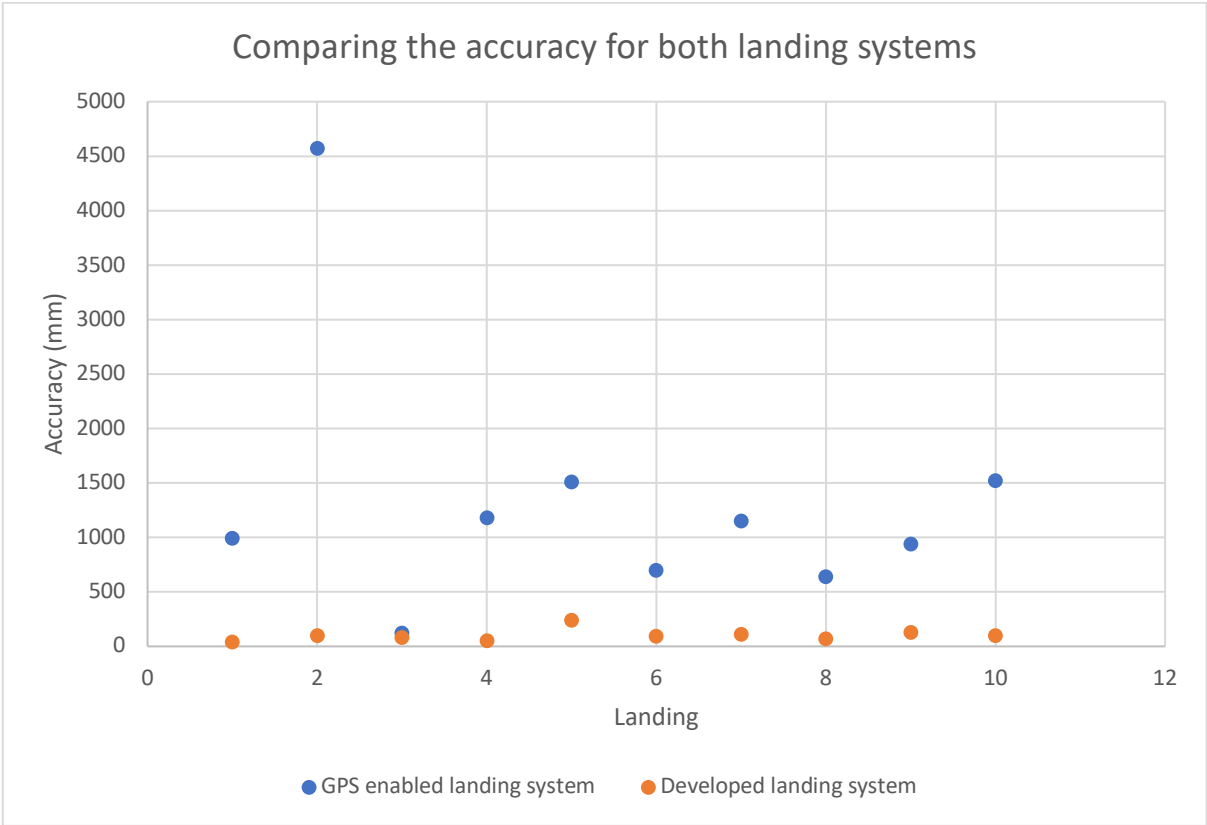


Figure 7.9 The plot for the accuracy of landing the aircraft for both landing systems

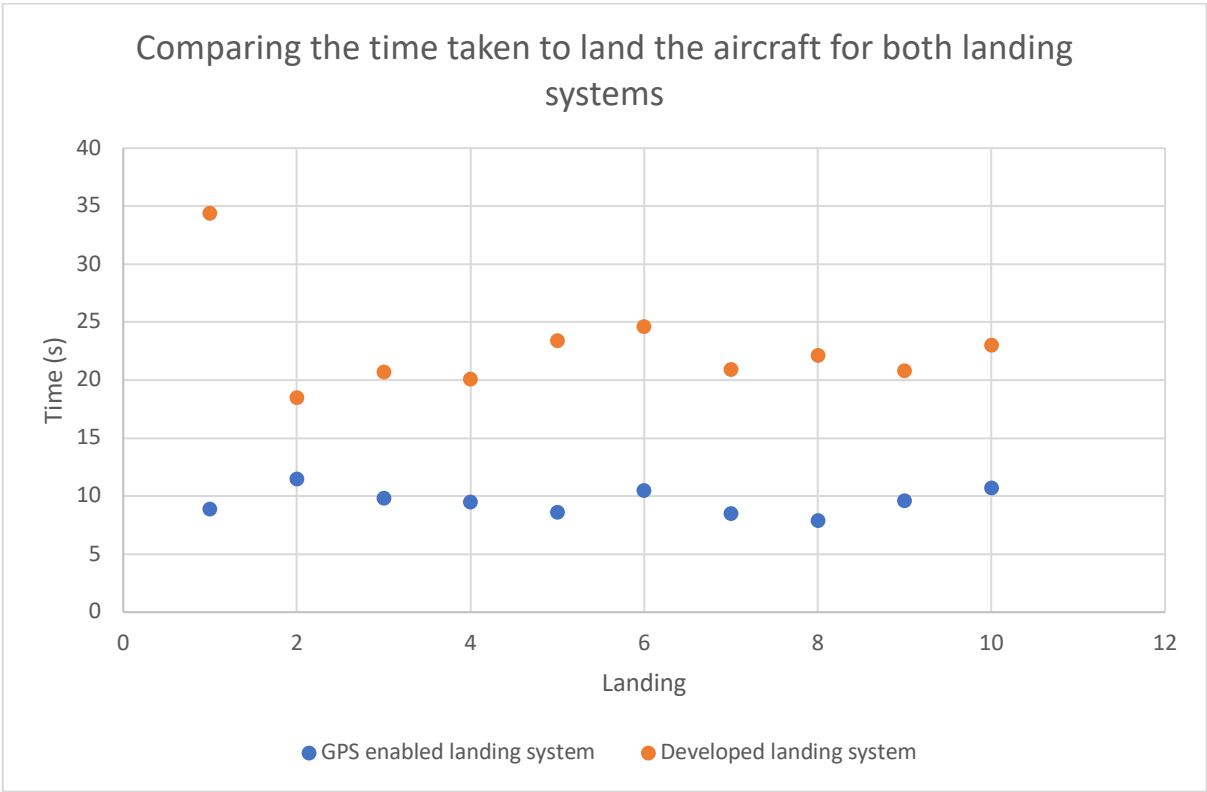


Figure 7.10: The plot for the time taken to land the aircraft for both landing systems

Referring to Figure 7.9, it can clearly be seen that the developed landing system had a better accuracy than the GPS enabled landing system. However, referring to Figure 7.10, the developed landing system did generally take longer to land the aircraft compared to the GPS enabled system. Since the purpose of developing a landing system was to improve on the accuracy of the overall landing of the aircraft, it can be said that the developed landing system outperformed the GPS enabled landing system in this regard.

If, in a real life scenario, an autonomous docking station (that can autonomously charge an aircraft) were to be developed for the aircraft, the aircraft would need to land accurately on the platform of the docking station to allow for it to be charged autonomously. For this reason, the accuracy of landing the aircraft would be more important than the time taken for it to land. Further research and developments in reducing the time taken to land the aircraft would result in a greater efficiency of the landing system.

7.2 Human Detection System

This section will discuss the following:

- Comparing Darknet's YOLO version 3 to Darknet's YOLO tiny version 3.
- Differentiating between a human and another object.
- Correcting the calculated distance.

When it came to testing the human detection system, it soon became apparent that the processing power of the onboard Odroid XU4 computer would be insufficient to run Darknet's YOLO. Therefore, all testing of the human detection system was performed on the desktop computer, where it used the computer's processor to perform Darknet's necessary calculations (as stated in section 6.1). A solution to the insufficient power of the onboard computer is discussed in section 7.3.

7.2.1 Comparing Darknet's YOLO version 3 to Darknet's YOLO Tiny Version 3

Since there are multiple versions of Darknet's YOLO available, a decision was made to compare the version 3 to the tiny version 3. The tiny version operates in the same way as the standard version. However, the tiny version gives less accurate results compared to the standard version due to its using less processing power from the computer to detect objects.

To compare the two versions, a test was performed where a pre-recorded video was published to a topic (by using the `video_stream_opencv` package) and both versions of Darknet's YOLO run independently to detect any objects in the published topic. The results for the Darknet's YOLO version 3 can be found in Figure 7.11 and the results for Darknet's YOLO tiny version 3 can be found in Figure 7.12. Both figures show:

- The bounding box formed around the detected person.
- The left terminal showing the frames per second (FPS) and the probability of the detected person.
- The right terminal showing the bounding box information for the detected object.



Figure 7.11: The detected person using Darknet's YOLO version 3



Figure 7.12: The detected person using Darknet's YOLO tiny version 3

It can be seen in Figure 7.11 that the standard version 3 managed to detect a person with a probability of 91%, which is higher than the 70% obtained for the tiny version 3 in Figure 7.12. This means that the standard version 3 is able to detect objects better than the tiny version 3. However, the 0.1 FPS in Figure 7.11 is lower than the 1.0 FPS obtained in Figure 7.12.

From the above, it can be deduced that the standard version would take roughly ten times longer to detect an object compared to the tiny version (for example, the standard version would take roughly ten seconds to detect a human compared to the one second obtained from the tiny version). For this reason, if the system were to be implemented on an onboard computer, the tiny version would definitely be the preferred choice. However, you would not achieve the best detection results. This can also be seen in Figure 7.13, where the tiny version incorrectly thought that the person in the video feed was a bird.

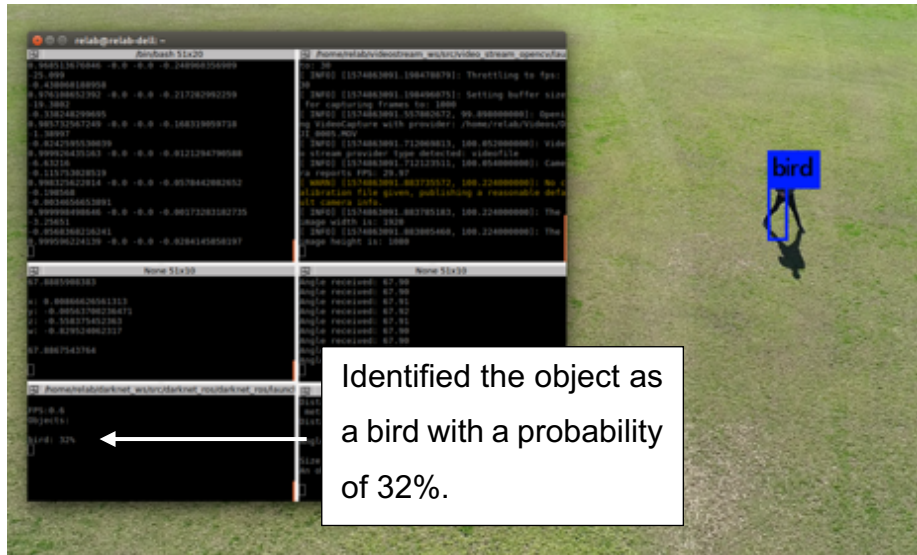


Figure 7.13: Darknet's YOLO tiny version 3 incorrectly identified the person as a bird

7.2.2 Differentiating Between a Human and Another Object

If the complete system were to be implemented in a real-world scenario, the vision system would need to be able to differentiate between different moving objects and be programmed only to follow a person. To prove that this was possible, a test was performed where a video was recorded of a person walking a dog on a field. The pre-recorded video was once again published to a topic by using the ROS video_stream_opencv package. For this test, the standard YOLO version 3 was used. Referring to Figure 7.14, the two detected objects can be seen, where a 100% probability for each object was obtained.

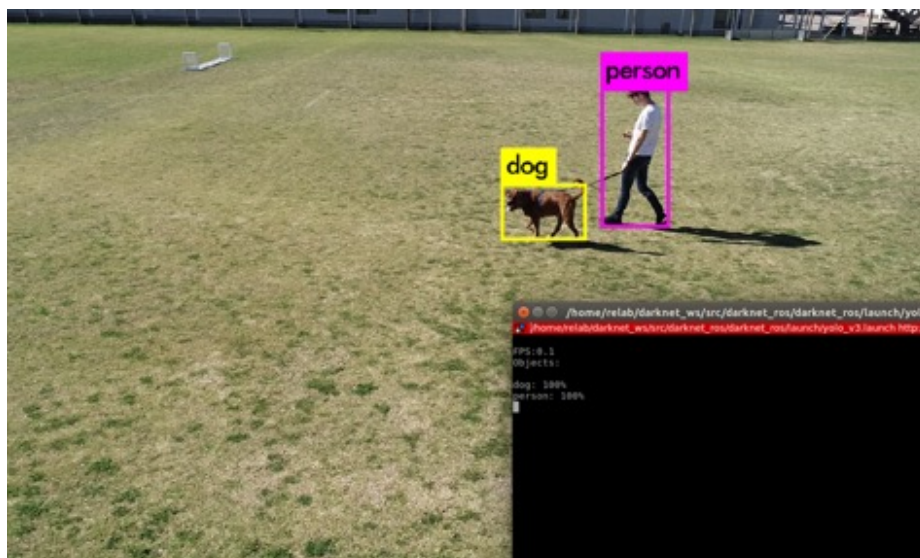


Figure 7.14: A person and a dog was detected by the vision system

7.2.3 Correcting the Calculated Distance

During the simulation process for the human detection system, an error was detected in the calculating the distance the aircraft needed to travel towards the human. Originally, when the “follow” command was inserted in the talker_autoland.cpp node, the aircraft determined the ground distance in the beginning. However, when the aircraft rotated to centre the person in the video frame, the Ymax of the bounding box of the person was reduced. This resulted in the person being further away than originally anticipated.

To demonstrate this, a person was told to stand still, walk a few steps horizontally to his left and then stand still, where the DJI drone would rotate itself to centre the person in the video frame. Figure 7.15 shows the original position of the person, Figure 7.16 is after the person had walked a few steps to their left and Figure 7.17 is a screenshot of the person once the aircraft had rotated itself to centre the person in the video frame. It must be noted that the standard YOLO version 3 was used to obtain a more accurate result.

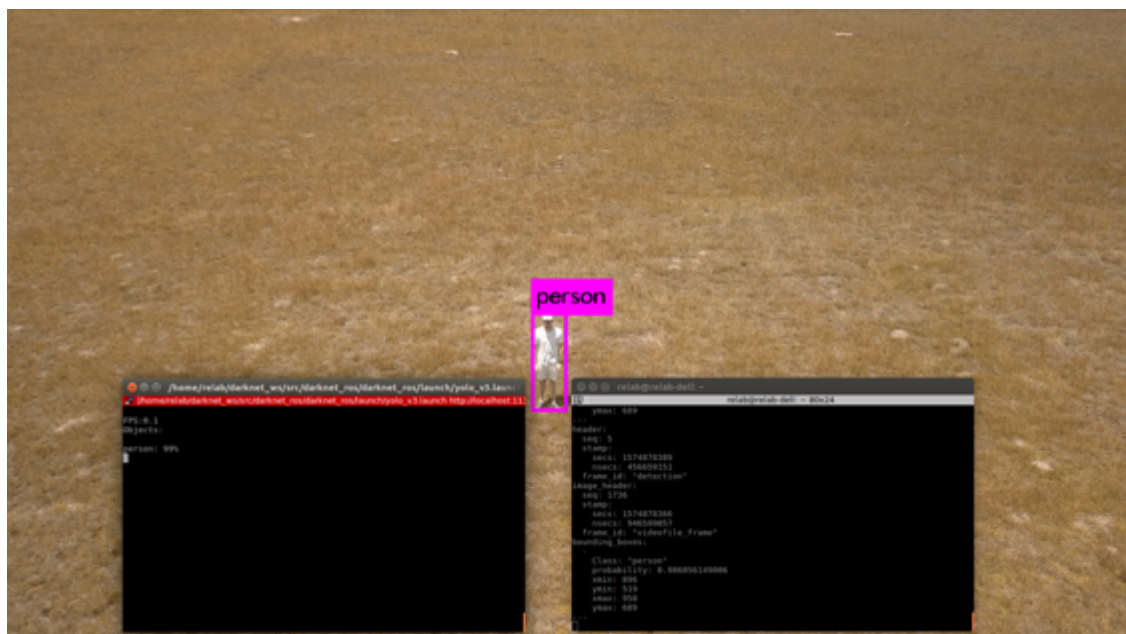


Figure 7.15: The bounding box information of the person in their original position

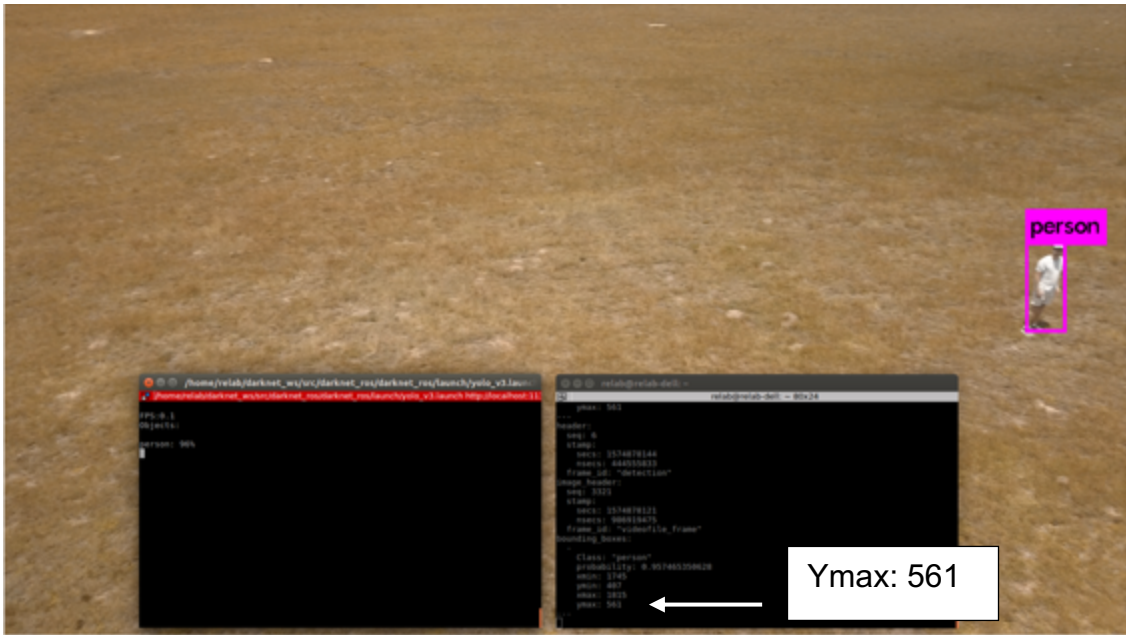


Figure 7.16: The bounding box information of the person after they had walked a few steps to their left

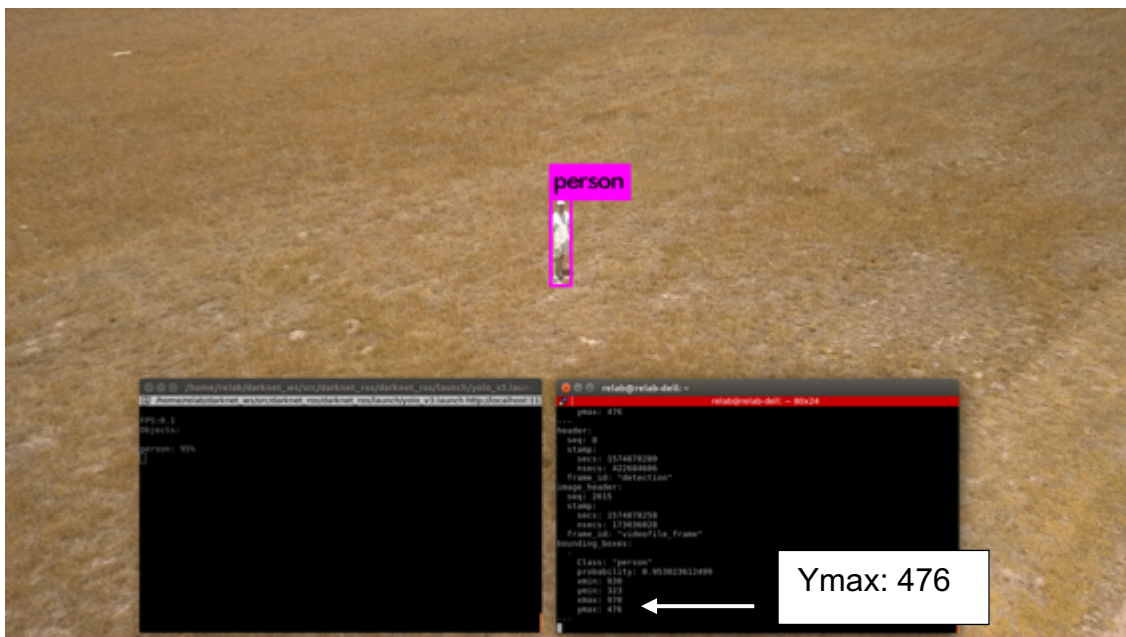


Figure 7.17: The bounding box information of the person after the DJI had rotated to centre the person in the video frame

When comparing Figure 7.16 and Figure 7.17, it can be seen that Figure 7.17's Ymax was smaller than that of Figure 7.16. There were two potential solutions to resolve this problem:

1. Redesign the algorithm for the human detection system only to calculate the ground distance once the aircraft had rotated.
2. Develop an equation to insert into the algorithm to calculate the potential future Ymax (after the aircraft has rotated), depending on the current position of the bounding box. This calculation would be added to the existing ground distance originally calculated, resulting in a slightly greater ground distance.

Due to time constraints, it was decided to develop an equation. To achieve this, the process followed was:

1. The DJI was flown and hovered at an altitude of ten metres. The DJI recorded a video of this entire process.
2. A person was required to stand in the centre of the video frame. This was regarded as the home position.
3. The person walked five steps to his left and remained stationary.
4. The aircraft rotated to centre the person in the video frame.
5. The aircraft would then rotate back to its original position to face the home position.
6. The process was repeated another four times where the person walked another five steps and the aircraft was rotated.
7. The video was process through the standard Darknet YOLO version 3, where the bounding box values for each position were recorded.

This entire process was repeated three times (resulting in three tests performed), where the Ymax value of the home position of the person was different each time. The three tests were recorded in Excel, which can be found in Appendix 7.4 . The numbers in the position column refer to when the person took a few steps horizontally and remained stationary. The letters in the position column refer to when the aircraft had rotated to centre the person in the video frame. This means that 1 correlates to A, 2 correlates to B, etc.. For example, Figure 7.15 would be regarded as the home position, Figure 7.16 would be regarded as position 1 and Figure 7.17 would be regarded as position A.

When determining the equation for each test, the average for the x values were used for the positions containing numbers, meaning:

$$X_{average} = \frac{X_{max} + X_{min}}{2} \quad (7.1)$$

The Xaverage was calculated for all positions, which can be found in the tables in Appendix 7.4. However only the Xaverage for the positions containing the numbers were used.

To calculate an equation, the Xaverage was plotted on a graph alongside the difference between the Ymax of when the person had taken a few steps and remained stationary and the Ymax of the home position.

For example, referring to the Test 1 table in Appendix 7.4, the Xaverage of position 1 was calculated (1208) and the Ymax of the position A (577) was subtracted from the Ymax of the home position (581). This resulted in a coordinate of (1208 ; 4).

Table 7.5, Table 7.6 and Table 7.7 were developed for the three tests. Graphs were plotted, based on these three tables, which can be referred to in Figure 7.18, Figure 7.19 & Figure 7.20. A linear equation was developed for each graph. These three equations were used in the algorithm to correct the ground distance.

Table 7.5: Test 1 points calculated

Position	Xaverage	Calculated Ymax Addition
home	960	0
1	1208	4
2	1448	4
3	1745.5	59
4	1839	83

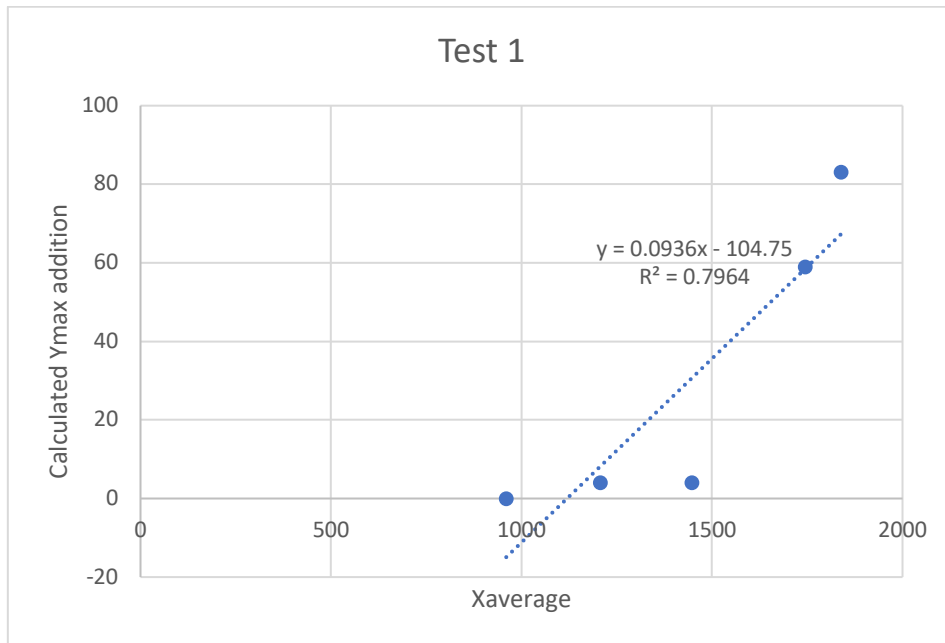


Figure 7.18: Graph plotted and equation developed for Test 1's data

Table 7.6: Test 2 points calculated

Position	Xaverage	Calculated Ymax Addition
home	960	0
1	1325.5	94
2	1579	175
3	1861	294

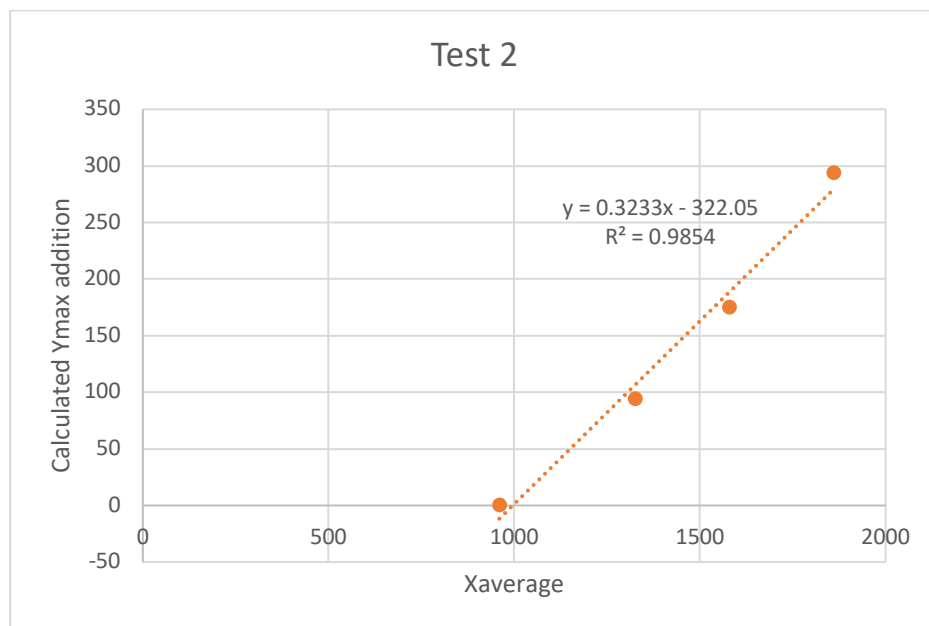


Figure 7.19: Graph plotted and equation developed for Test 2's data

Table 7.7: Test 3 points calculated

Position	Xaverage	Calculated Ymax Addition
home	960	0
1	1200.5	10
2	1352.5	48
3	1555.5	90
4	1748	135

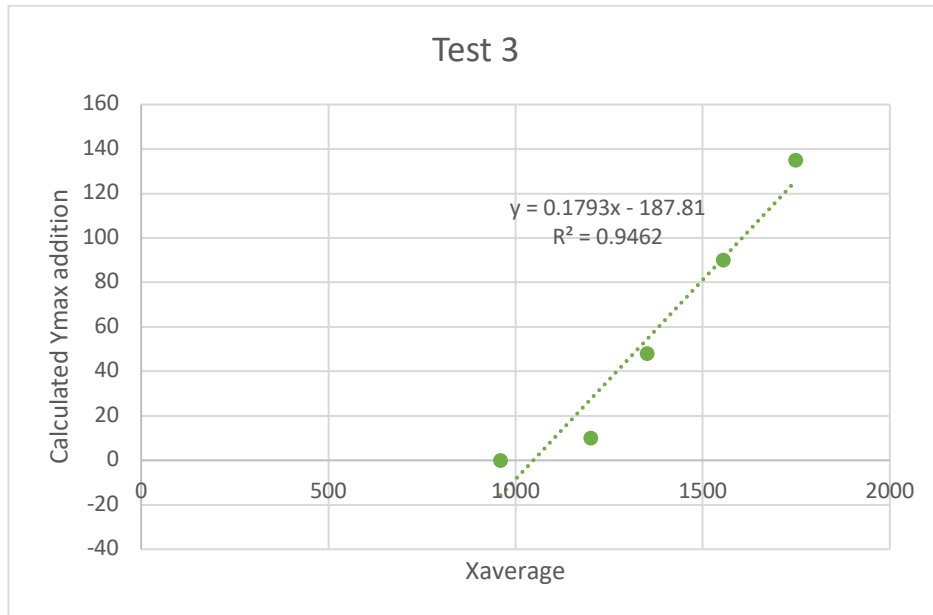


Figure 7.20: Graph plotted and equation developed for Test 3's data

It must be noted that for the three tests, the Xaverage was only greater than the midpoint of 960 pixels and was never calculated for the Xaverage to be below 960 pixels. This meant that if the Xaverage was smaller than 960, the difference between the two was added to the midpoint of 960 pixels to obtain a new Xaverage. For example, say the Xaverage was 925, the following calculations were implemented:

$$X_{difference} = X_{midpoint} - X_{average}$$

$$X_{difference} = 960 - 925$$

$$X_{difference} = 35$$

Therefore,

$$X_{newaverage} = X_{midpoint} + X_{difference}$$

$$X_{newaverage} = 960 + 35$$

$$X_{newaverage} = 995$$

This $X_{newaverage}$ would be inserted into one of the three equations developed.

When determining which equation to use, a range was used based on the Y_{max} value.

The ranges were:

- If Y_{max} was smaller than 500 pixels ($Y_{max} < 500$), then test 3's equation was used.
- If Y_{max} was greater or equal to 500 pixels and Y_{max} was smaller than 750 pixels ($500 \leq Y_{max} < 750$), then test 1's equation was used.
- If Y_{max} was greater or equal to 750 pixels ($Y_{max} \geq 750$), then test 2's equation was used.

Using one of these three equations does not replace the originally calculated ground distance, but is added to the originally calculated ground distance, resulting in a greater overall ground distance needed to be travelled by the aircraft.

7.3 Improvements to the System

Due to the system containing many components, there was bound to be a few of them that needed to be upgraded. From the beginning of the research, the idea was to keep the overall project costs to a minimum. That is why numerous second-hand components were used for the system. The following improvements could be implemented on the aircraft:

- Since the Odroid XU4 companion computer was not capable of being used for the human detection system, a more powerful computer is required. The ideal solution to this problem is to use an Nvidia Jetson TX2 module attached to a carrier board. This module is ideal because Darknet's YOLO was developed on the CUDA framework, which works best with Nvidia graphics cards. The purpose of the carrier board is to provide the module with IOs (inputs/outputs). This choice of computer will drastically improve on the FPS that was achieved in this research.
- Due to the shortage of space on the fuselage, some of the peripherals and the companion computer had to be placed on their sides to fit all of the components onto the aircraft. This resulted in some of the components not being tightly

fitted onto the aircraft. To improve on this, the aircraft's fuselage should be redesigned to house all the components in a neat, compact system.

- The 6600 mAh capacity of the LiPo batteries were emptied between roughly five to seven minutes due to the batteries needing to power numerous components. To avoid this, bigger batteries could be used or more added to the system, where they could be assigned to power specific components (e.g. one battery could power the companion computer and the peripherals connected to the computer).
- If the system were to be used in a real-world scenario, the webcam would need to be replaced with a camera for industrial use. Thermal cameras and multispectral cameras should be considered in this regard.

7.4 Conclusion

The numerous tests flights that were executed on the aircraft were discussed, as well as the various ways of flying the aircraft, albeit manually or autonomously. In the chapter, other ways of controlling the aircraft, such as injecting new waypoints for the aircraft to fly towards by using ROS as well as using an Arduino to send the aircraft on a mission were mentioned. The two landing systems of the aircraft were discussed and compared, highlighting the developed landing system outperforming the GPS enabled landing system. Finally the human detection system was discussed, comparing Darknet's YOLO version 3 to Darknet's YOLO tiny version 3, revealing that the system was able to differentiate between a human and a dog as well as correct the calculated ground distance. Improvements to the system were suggested, which could potentially allow the aircraft to be used in detecting and following humans. The conclusion follows in the next chapter, which will conclude the entire research paper.

Chapter 8: Conclusion

The purpose of this chapter is to conclude the dissertation, highlighting the developed multirotor aircraft as well as the simulations and testing performed. The research contributions and the original hypothesis will be discussed. The chapter will also include recommendations for future research.

8.1 Conclusion

This dissertation demonstrated the vast development of multirotor aircraft in today's era. One of the topics that was investigated was the latest crime statistics of South Africa, which revealed that a new method or device was required to help reduce the overall crime in the country. This led to the development of this research.

The focus was to provide the aircraft with another method of landing itself, which did not need to make use of the expensive RTK system that was currently available. This led to developing a vision system for the aircraft. The vision system was used for the developed landing system as well as for the human detection system. The vision system comprised a Logitech C920 webcam (which was mounted on a Tarot gimbal) that was connected to a Odroid XU4 companion computer running ROS. The reason for ROS being used on the computer was to integrate the entire system, allowing the vision system to communicate with the aircraft, informing the aircraft to re-orientate and reposition itself based on what the vision system detected.

The developed landing system consisted of detecting an ArUco marker placed on the ground. When instructed to land, the aircraft repositioned itself above the marker, adjusted its orientation to align itself correctly with the marker and began its descent, eventually landing on the ArUco marker. The human detection system consisted of detecting a human by making use of Darknet's YOLO version 3 and Darknet's YOLO tiny version 3.

Both the landing system and the human detection system were developed by making use of a SITL simulation (otherwise known as a Software in the Loop simulation). This SITL was run alongside Gazebo, allowing for a model in Gazebo to be manipulated

by making use of a ROS node. This enabled all the ROS nodes to be tested in Gazebo first, before testing them on the actual aircraft.

After successfully testing the landing system in Gazebo, the system was implemented onto the actual aircraft, where physical testing took place. After making adjustments in the algorithms to force the aircraft to touch the ground, the aircraft successfully managed to land. However, the aircraft was never able to disarm due to an issue in the PX4 firmware. The developed landing system was then compared to the aircraft's GPS based landing system, where ten landings were performed using each system. The developed landing system obtained a better accuracy than the GPS enabled landing system. However, the time taken to land the aircraft using the developed landing system took longer than the GPS enabled landing system.

The human detection system was only tested in the simulation and not on the actual aircraft due to the Odroid XU4 companion computer providing insufficient processing power. This limited the amount of testing possible for the system.

There were numerous tests performed on the aircraft, showing that the aircraft was successfully able to be flown manually and autonomously. The main reason that this was possible was due to the open-source Pixhawk flight controller running PX4 firmware being used. This choice in firmware allowed for the aircraft to be able to be integrated with ROS. The decision to use ROS for this research helped integrate the entire system with ease.

There were a few limitations that were experienced when doing the research. The first was that the aircraft was only flown in light wind conditions. So, the results recorded for the GPS based landing system and the developed landing system do not reflect how the aircraft would perform in stronger wind conditions. The second limitation was that the human detection system was only tested on two humans and one dog. This system should be tested further on a larger range of subjects, varying in size and height.

Looking at the research contributions mentioned in section 1.7, all of the points mentioned were obtained. A vision system was developed that could detect ArUco

markers and detect objects within the video frame. A multirotor aircraft was developed that could be flown manually and autonomously. The developed landing system was able to help improve on the accuracy when the aircraft landed. However, this improvement in accuracy resulted in the aircraft taking longer to land. If the developed landing system was improved by performing further research in this regard, the time taken to land the aircraft could be reduced substantially, resulting in a more efficient landing system. The developed human detection system was able to be used to manipulate the aircraft during flight. However, this human detection system was not implemented on the aircraft due to the aircraft's onboard computer not having sufficient processing power to execute the said detection system. This could easily be addressed by using a more powerful computer. Therefore, further research should be performed in testing the human detection system on the aircraft. Finally, the vision system and the aircraft were integrated by ROS. This integration will assist other students in performing research in a similar field as the necessary foundation for their research has been laid.

Referring to the original hypothesis a hardware and system architecture was designed and a digital simulation environment was used and implemented. An alternative landing system was successfully developed and tested on the actual aircraft. A human detection was successfully developed, however it was never tested on the actual aircraft. From these points, it can be said that a vision based multirotor aircraft can in fact be developed to be used in the security industry.

8.2 Recommendations for Future Research

During this research, a number of ideas came to mind where future research could be implemented:

- Develop a docking station that could charge and protect the multirotor aircraft when not being used.
- Design a waterproof fuselage so that the aircraft could house all the components into a neat compact system, allowing the multirotor aircraft to be flown in raining or windy conditions.
- Develop a battery management system to improve on the flight time of the aircraft.

- Develop a communication system to allow the aircraft to be controlled and the video feed of the aircraft to be viewed from a control room.
- Improve on the human detection system.

Bibliography

- Aerial Robotics, 2018. aruco_detection_gazebo. [Online] Available at: https://github.com/AerialRobotics-IITK/aruco_detection_gazebo [Accessed 23 November 2019].
- Aerial Robotics, 2019. Github offb_node.cpp. [Online] Available at: https://github.com/AerialRobotics-IITK/aruco_detection_gazebo/blob/master/offboard/src/offb_node.cpp#L4 [Accessed 31 January 2019].
- Allen, C., 2014. How a Quadcopter Works. [Online] Available at: http://ffden-2.phys.uaf.edu/webproj/212_spring_2014/Clay_Allen/clay_allen/works.html [Accessed 5 April 2017].
- Amazon, 2019. FPVKing 6M GPS Module Built-in Compass +Black GPS Folding Antenna Mount Holder for APM2.6 APM2.8 Pixhawk Flight Controller. [Online] Available at: <https://www.amazon.com/FPVKing-Compass-Folding-Pixhawk-Controller/dp/B077DC7WDB> [Accessed 6 November 2019].
- AnimMotion, 2019. Unit Quaternions vs Euler Angles: The Pros. [Online] Available at: <http://peyman-mass.blogspot.com/2013/04/unit-quaternions-vs-euler-angles.html> [Accessed 11 November 2019].
- ArduPilot, 2016. ArduPilot. [Online] Available at: <http://ardupilot.org/about> [Accessed 29 October 2019].
- ArduPilot, 2019. Choosing a Ground Station. [Online] Available at: <http://ardupilot.org/copter/docs/common-choosing-a-ground-station.html> [Accessed 24 October 2019].
- ArduPilot, 2019. Connect ESCs and Motors. [Online] Available at: <https://ardupilot.org/copter/docs/connect-escs-and-motors.html> [Accessed 30 November 2019].
- ArduPilot, 2019. Copter Flight Modes. [Online] Available at: <http://ardupilot.org/copter/docs/flight-modes.html> [Accessed 9 November 2019].

Aviation History, 2015. Airfoils and Lift. [Online] Available at: <http://www.aviation-history.com/theory/airfoil.htm> [Accessed 17 May 2017].

Banggood, 2019. Holybro Pix32 Pixhawk PX4 2.4.6 Flight Controller & Buzzer & Power Module with XT60 Set. [Online] Available at: https://www.banggood.com/Holybro-Pix32-Pixhawk-PX4-2.4.6-Flight-Controller-Buzzer-Power-Module-with-XT60-Set-p-1310396.html?gmcCountry=ZA¤cy=ZAR&createTmp=1&utm_source=googleshopping&utm_medium=cpc_union&utm_content=xibei&utm_campaign=xibei-ssc-za-a [Accessed 22 November 2019].

Banzi, M. & Shiloh, M., 2014. Make: Getting Started With Arduino. 3rd ed. Sebastopol: Maker Media Inc. Arduino, 2017. Arduino. [Online] Available at: <https://www.arduino.cc/en/> [Accessed 6 November 2017].

Build Your Own Drone, 2019. DJI F550 FLAME WHEEL E305 ARF KIT V2. [Online] Available at: <https://www.buildyourowndrone.co.uk/dji-f550-flame-wheel-e305-artf-v2> [Accessed 6 November 2019].

Business Tech, 2017. South Africa ranks among the most dangerous countries in the world - it's costing us. [Online] Available at: <https://businesstech.co.za/news/lifestyle/200044/south-africa-ranks-among-the-most-dangerous-countries-in-the-world-and-its-costing-us/> [Accessed 7 February 2018].

CHRobotics, 2019a. Understanding Euler Angles. [Online] Available at: <http://www.chrobotics.com/library/understanding-euler-angles> [Accessed 11 November 2019].

CHRobotics, 2019b. Understanding Quaternions. [Online] Available at: <http://www.chrobotics.com/library/understanding-quaternions> [Accessed 12 November 2019].

Euclidean Space, 2017. Maths - Quaternion Code. [Online] Available at: <https://www.euclideanspace.com/maths/algebra/realNormedAlgebra/quaternions/code/index.htm> [Accessed 21 May 2019].

GetFPV, 2019. FrSky Taranis Q X7 2.4GHz 16CH Transmitter (Black). [Online] Available at: <https://www.getfpv.com/frsky-taranis-q-x7-2-4ghz-16ch-transmitter-black.html> [Accessed 7 November 2019].

GetFPV, 2019. Pixhawk 2.1 Standard Set. [Online] Available at: <https://www.getfpv.com/pixhawk-2-1-standard-set.html> [Accessed 7 November 2019].

GitHub, 2017. Sending waypoint info by using mavros services. [Online] Available at: <https://github.com/mavlink/mavros/issues/837> [Accessed 8 October 2018].

Hardkernel, 2019a. ODROID-XU4. [Online] Available at: <https://www.hardkernel.com/shop/odroid-xu4-special-price/> [Accessed 7 November 2019].

Hardkernel, 2019b. 32GB eMMC Module XU4 Linux. [Online] Available at: <https://www.hardkernel.com/shop/32gb-emmc-module-xu4-linux/> [Accessed 10 November 2019].

Hasko, D., 2016. Generate PPM signal with Arduino. [Online] Available at: <https://quadmeup.com/generate-ppm-signal-with-arduino/> [Accessed 10 December 2019].

Heli Engadin, 2019. Pixhawk 2.1 Standard Set & Here+ V2 RTK GNSS Combo. [Online] Available at: <https://www.heliengadin.com/products/pixhawk2-1-standard-set-here-rtk-gnss-combo> [Accessed 7 November 2019].

Hobby-Miracle, 2019. APC 10 x 4.7 Slow Flyer Propeller. [Online] Available at: http://hobby-miracle.com/index.php?route=product/product&product_id=2423 [Accessed 14 November 2019].

HobbyKing, 2019. PROPDRIVE v2 2826 1000KV Brushless Outrunner Motor. [Online] Available at: https://hobbyking.com/en_us/propdrive-v2-2826-1000kv-brushless-outrunner-motor.html [Accessed 6 November 2019].

- Hobbywing, 2015. UBEC 8A (2-3S). [Online] Available at: http://www.hobbywing.com/goods.php?id=378&filter_attr=5758.0 [Accessed 8 November 2019].
- How Things Fly, 2005. The Four Forces. [Online] Available at: <http://howthingsfly.si.edu/forces-flight/four-forces> [Accessed 10 June 2017].
- Howse, J., Joshi, P. & Beyeler, M., 2016. OpenCV: Computer Vision Projects with Python. Birmingham: Packt Publishing.
- IGI Global, 2019. What is object tracking. [Online] Available at: <https://www.igi-global.com/dictionary/moving-object-detection-and-tracking-based-on-the-contour-extraction-and-centroid-representation/20697> [Accessed 25 September 2019].
- Institute for Economics & Peace, 2017. Global Peace Index 2017, pp. 1-140.
- ISC West, 2019. Robotic Aerial Security. [Online] Available at: <https://www.iscwest.com/en/Exhibitors/5630991/Nightingale-Security/Products/1519649/Robotic-Aerial-Security> [Accessed 28 October 2019].
- Kadamatt, V., 2017. How quadcopters work & fly: An intro to multirotors. [Online] Available at: <http://www.droneybee.com/how-quadcopters-work/> [Accessed 25 April 2017].
- Koubaa, A., Allouch, A. & Alajlan, M., 2015. Micro Air Vehicle Link (MAVLink) in a Nutshell: A Survey. Journal of Latex Class Files, 14(8), p. 3.
- LambDrive, 2016. Pixhawk Family. [Online] Available at: <https://www.lambdrive.com/depot/Robotics/Controller/PixhawkFamily/index.html> [Accessed 28 October 2019].
- Logitech Apps, 2019. Logitech C920 Pro Webcam Review. [Online] Available at: <https://logitech-apps.com/logitech-c920/> [Accessed 22 July 2019].
- Logitech, 2019. C920 HD PRO WEBCAM. [Online] Available at: <https://www.logitech.com/en-hk/product/hd-pro-webcam-c920> [Accessed 8 November 2019].

- Lopez, J. L. S., 2016. Aruco Visual Markers for Gazebo Simulator. [Online] Available at: https://github.com/joselusl/aruco_gazebo [Accessed 20 February 2019].
- MAVLink, 2019. MAVLink Developer Guide. [Online] Available at: <https://mavlink.io/en/> [Accessed 19 September 2019].
- MAVLink, 2019. Packet Serialization. [Online] Available at: <https://mavlink.io/en/guide/serialization.html> [Accessed 14 October 2019].
- Mostashiri, N. & Dhupia, J. S., 2018. A novel spatial mandibular motion-capture system based on planar fiducial markers. IEEE Sensors - Engineering.
- NASA, 2015. What is Drag?. [Online] Available at: <https://www.grc.nasa.gov/www/k-12/airplane/drag1.html> [Accessed 11 June 2017].
- Nightingale Security, 2019. Nightingale Security. [Online] Available at: <https://www.nightingalesecurity.com> [Accessed 29 October 2019].
- OpenCV, 2019. Detection of ArUco Markers. [Online] Available at: https://docs.opencv.org/trunk/d5/dae/tutorial_aruco_detection.html [Accessed 22 October 2019].
- OpenCV, 2019. OpenCV About. [Online] Available at: <https://opencv.org/about/> [Accessed 12 November 2019].
- Parker, G., 2018. Drone vs UAV - What is the difference. [Online] Available at: <https://wiki.ezvid.com/m/drone-vs-uav-what-is-the-difference-2FJYpSrUkP-> [Accessed 15 July 2018].
- Pixhawk, 2019. Pixhawk. [Online] Available at: <https://pixhawk.org> [Accessed 29 October 2019].
- PorcupineRC, 2019. FrSky X8R 8/16CH SBUS Telemetry Receiver (ACCST Compatible). [Online] Available at: http://www.porcupinerc.com/FrSky-X8R-816CH-SBUS-Telemetry-Receiver-ACCST-Compatible_p_544.html [Accessed 7 November 2019].

PX4, 2018. PX4 Autopilot. [Online] Available at: <https://px4.io> [Accessed 29 October 2019].

PX4, 2019a. Pixhawk 1 & Turnigy TGY-IA6C 2.4Ghz Receiver. [Online] Available at: <https://discuss.px4.io/t/pixhawk-1-turnigy-tgy-ia6c-2-4ghz-receiver/9190> [Accessed 5 November 2019].

PX4, 2019b. Pixhawk 1 Flight Controller. [Online] Available at: https://docs.px4.io/v1.9.0/en/flight_controller/pixhawk.html [Accessed 8 November 2019].

PX4, 2019c. Flight Modes. [Online] Available at: https://docs.px4.io/v1.9.0/en/flight_modes/ [Accessed 9 November 2019].

PX4, 2019d. RTK GPS. [Online] Available at: https://docs.px4.io/v1.9.0/en/gps_compass/rtk_gps.html [Accessed 14 November 2019].

PX4, 2019e. Precision Landing. [Online] Available at: https://docs.px4.io/v1.9.0/en/advanced_features/precland.html [Accessed 14 November 2019].

PX4, 2019f. MAVROS offboard control example. [Online] Available at: https://dev.px4.io/v1.9.0/en/ros/mavros_offboard.html [Accessed 31 January 2019].

PX4, 2019g. Gazebo Simulation. [Online] Available at: <https://dev.px4.io/v1.9.0/en/simulation/gazebo.html> [Accessed 23 November 2019].

PX4, 2019h. Using Vision or Motion Capture Systems for Position Estimation. [Online] Available at: https://dev.px4.io/v1.9.0/en/ros/external_position_estimation.html [Accessed 3 December 2019].

Quadcopter Arena, 2017. The History of Drones and Quadcopters. [Online] Available at: <http://quadcopterarena.com/the-history-of-drones-and-quadcopters/> [Accessed 20 July 2018].

- Quaternion Simulator, 2019. Quaternions. [Online] Available at: <https://quaternions.online> [Accessed 21 May 2019].
- Quigley, M., Gerkey, B. & Conley, K., n.d. ROS: an open-source Robot Operating System. California, s.nop.
- Raspberry Pi, 2017. VNC (Virtual Network Computing). [Online] Available at: <https://www.raspberrypi.org/documentation/remote-access/vnc/> [Accessed 10 November 2019].
- RC Groups, 2013. Afro 30a Multicopter ESC w/ SimonK. [Online] Available at: <https://www.rcgroups.com/forums/showthread.php?1948812-Afro-30a-Multicopter-ESC-w-SimonK> [Accessed 9 November 2019].
- Readytosky, 2019. Readytosky 3DRobotics Radio 433Mhz Telemetry Kit. [Online] Available at: http://www.readytosky.com/e_products/show/?383-Readytosky-3DRobotics-Radio-Telemetry-Kit-433Mhz-433Air-Module-383.html [Accessed 6 November 2019].
- Redmon, J., 2013. Darknet: Open Source Neural Networks in C. [Online] Available at: <https://pjreddie.com/darknet/> [Accessed 26 September 2019].
- Reid, J., 2016. Understanding Kv Ratings. [Online] Available at: <https://www.rotordronepro.com/understanding-kv-ratings/#outer-popup> [Accessed 21 November 2019].
- Robotics with ROS, 2017. Calibrating a Monocular Camera with ROS. [Online] Available at: <http://ros-developer.com/2017/04/23/camera-calibration-with-ros/> [Accessed 18 February 2019].
- ROS, 2014. aruco_ros. [Online] Available at: http://wiki.ros.org/aruco_ros [Accessed 20 February 2019].
- ROS, 2016a. usb_cam. [Online] Available at: http://wiki.ros.org/usb_cam [Accessed 20 February 2019].
- ROS, 2016b. rqt. [Online] Available at: <http://wiki.ros.org/rqt> [Accessed 20 February 2019].

- ROS, 2017. camera_calibration. [Online] Available at: http://wiki.ros.org/camera_calibration [Accessed 20 February 2019].
- ROS, 2018a. darknet_ros. [Online] Available at: http://wiki.ros.org/darknet_ros [Accessed 6 June 2019].
- ROS, 2018b. video_stream_opencv. [Online] Available at: http://wiki.ros.org/video_stream_opencv [Accessed 6 June 2019].
- ROS, 2018c. mavros. [Online] Available at: <http://wiki.ros.org/mavros> [Accessed 16 November 2019].
- ROS, 2019a. About ROS. [Online] Available at: <https://www.ros.org/about-ros/> [Accessed 3 September 2019].
- ROS, 2019b. Understanding Nodes. [Online] Available at: <http://wiki.ros.org/ROS/Tutorials/UnderstandingNodes> [Accessed 21 November 2019].
- ROS, 2019c. Is ROS for Me?. [Online] Available at: <https://www.ros.org/is-ros-for-me/> [Accessed 12 November 2019].
- Sewell, J.A., 2019. Vision-Based Autonomous Aircraft Payload Delivery System.
- Shipitko, O., 2017. 3D pose estimation algorithm for intelligent box picking of warehouse automation robot. Moscow, s.n.
- South African Police Service, 2019. SAPS Crimestats. [Online] Available at: <https://www.saps.gov.za/services/crimestats.php> [Accessed 11 October 2019].
- Sparkfun, 2015. Arduino Comparison Guide. [Online] Available at: <https://learn.sparkfun.com/tutorials/arduino-comparison-guide> [Accessed 17 June 2017].
- Sunflower Labs Inc., 2019. The Sunflower System. [Online] Available at: <https://sunflower-labs.com> [Accessed 28 October 2019].

Tarot, 2017. ZYX T-3D V User Manual. [Online] Available at: <http://www.tarotrc.com/Download/Detail.aspx?Lang=en&Id=7f048157-8b73-4bd7-b3de-5b5fd772d6e1> [Accessed 6 November 2019].

u-blox, 2015. NEO-M8, s.l.: s.n.

UAV Systems International, 2018. What is a gimbal?. [Online] Available at: <https://www.uavsystemsinternational.com/what-is-a-gimbal/> [Accessed 21 July 2018].

Unitek, 2019. USB3.0 4-Port Hub. [Online] Available at: <https://www.unitek-products.com/products/usb3-0-4-port-hub> [Accessed 7 November 2019].

Unmanned Tech, 2019. Gens ace 3300mAh 11.1V 25C 3S1P Lipo Battery Pack. [Online] Available at: <https://www.unmannedtechshop.co.uk/product/gens-ace-3300mah-11-1v-25c-3s1p-lipo-battery-pack/> [Accessed 8 November 2019].

Appendices

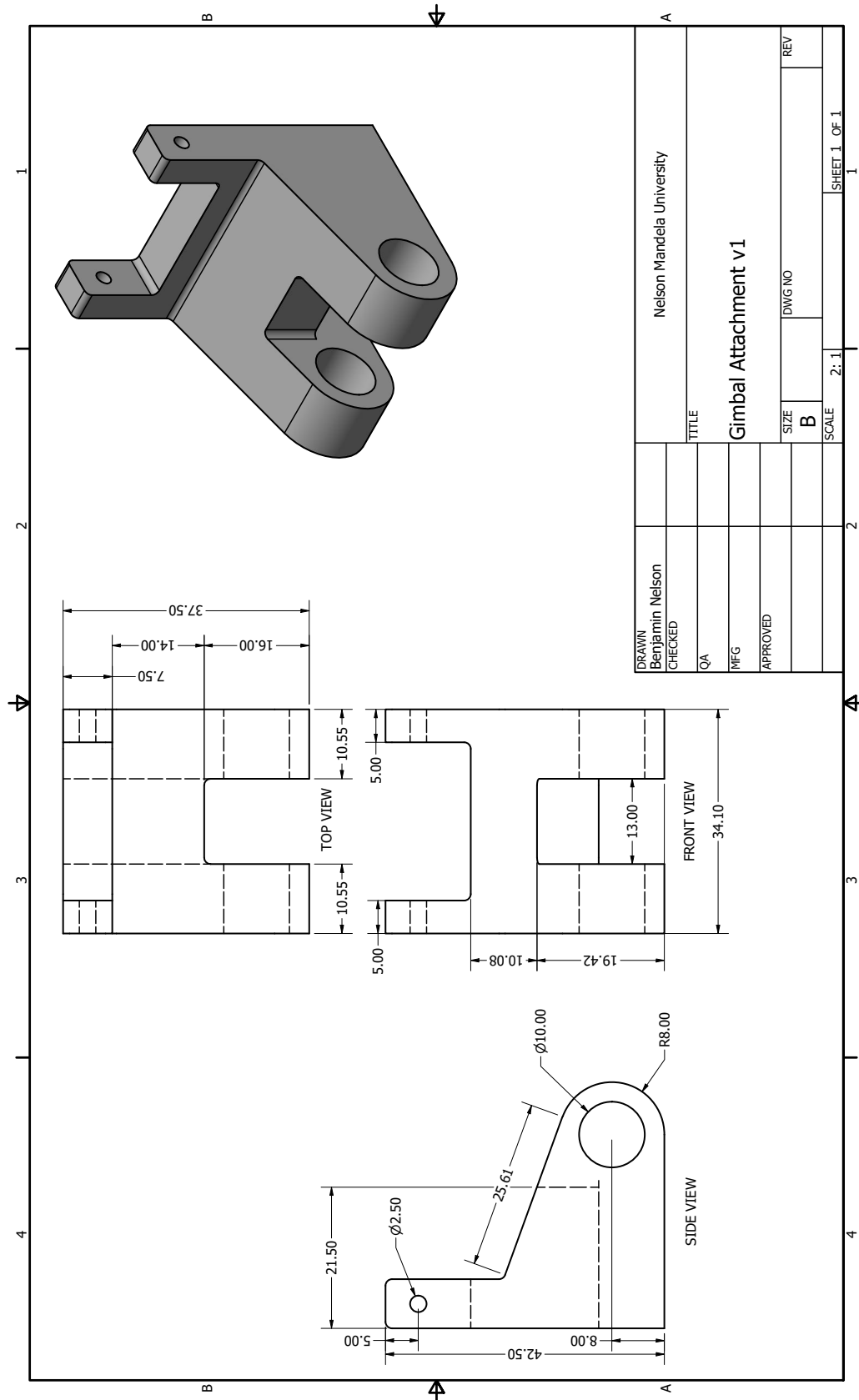
Appendix 1.1: South Africa's Crime Statistics

South Africa's Crime Statistics	
Crime Category	2018/2019
Murder	21 022
Sexual Offences	52 420
Attempted Murder	18 980
Assault with the intent to inflict grievous bodily harm	170 979
Common assault	162 012
Common robbery	51 765
Robbery with aggravating circumstances	140 032
Rape	41 583
Sexual Assault	7 437
Attempted Sexual Offences	2 146
Contact Sexual Offences	1 254
Carjacking	16 026
Robbery at residential premises	22 431
Robbery at non-residential premises	19 991
Robbery of cash in transit	183
Bank robbery	4
Truck hijacking	1 182
Arson	4 083
Malicious damage to property	113 089
Burglary at non-residential premises	71 224
Burglary at residential premises	220 865
Theft of motor vehicle and motorcycle	48 324
Theft out of or from motor vehicle	125 076
Commercial crime	83 823
Shoplifting	60 167

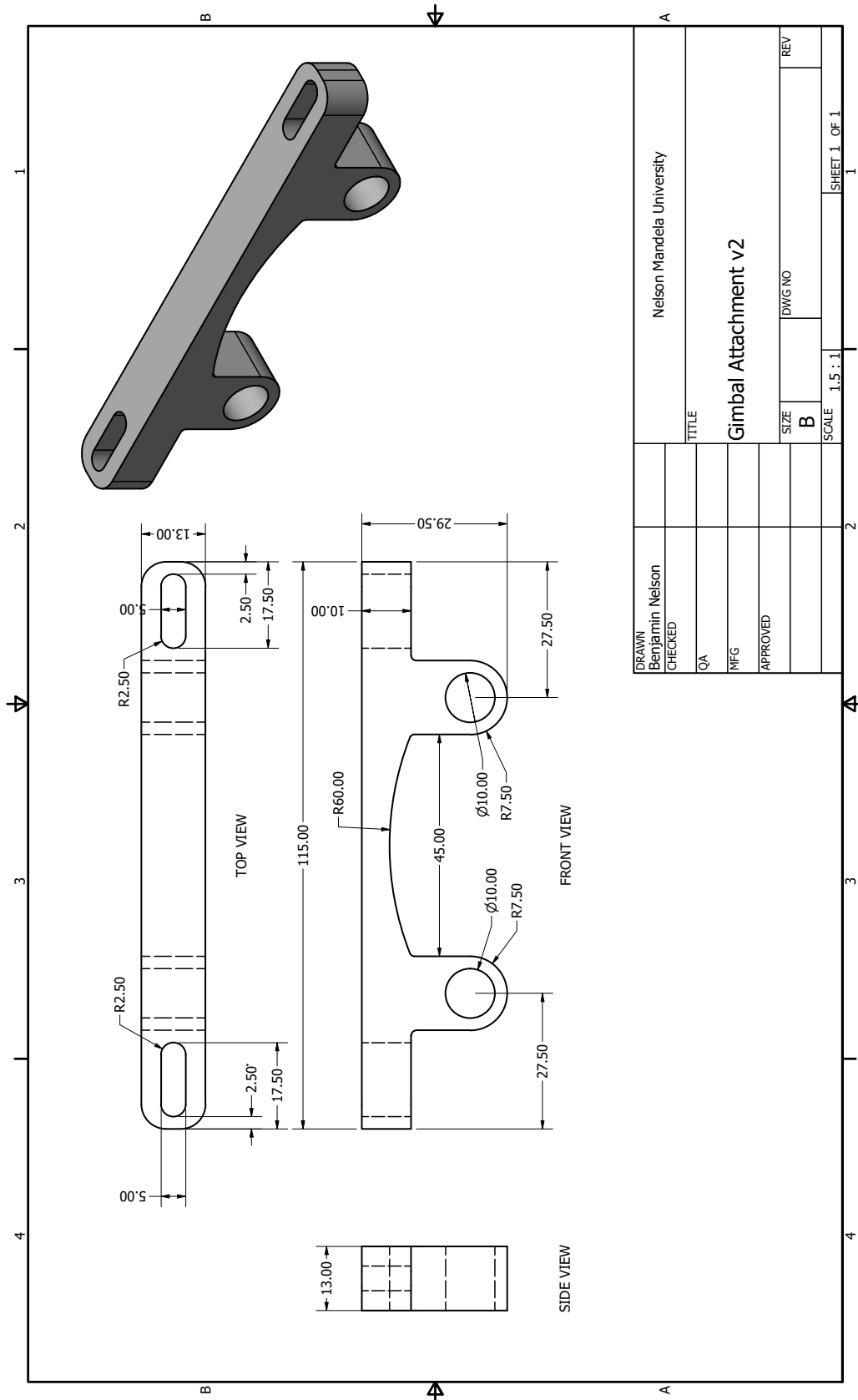
Appendix 2.1: Darknet's YOLO Dataset

person	elephant	wine glass	diningtable
bicycle	bear	cup	toilet
car	zebra	fork	tvmonitor
motorbike	giraffe	knife	laptop
aeroplane	backpack	spoon	mouse
bus	umbrella	bowl	remote
train	handbag	banana	keyboard
truck	tie	apple	cell phone
boat	suitcase	sandwich	microwave
traffic light	frisbee	orange	oven
fire hydrant	skis	broccoli	toaster
stop sign	snowboard	carrot	sink
parking meter	sports ball	hot dog	refrigerator
bench	kite	pizza	book
bird	baseball bat	donut	clock
cat	baseball glove	cake	vase
dog	skateboard	chair	scissors
horse	surfboard	sofa	teddy bear
sheep	tennis racket	pottedplant	hair drier
cow	bottle	bed	toothbrush

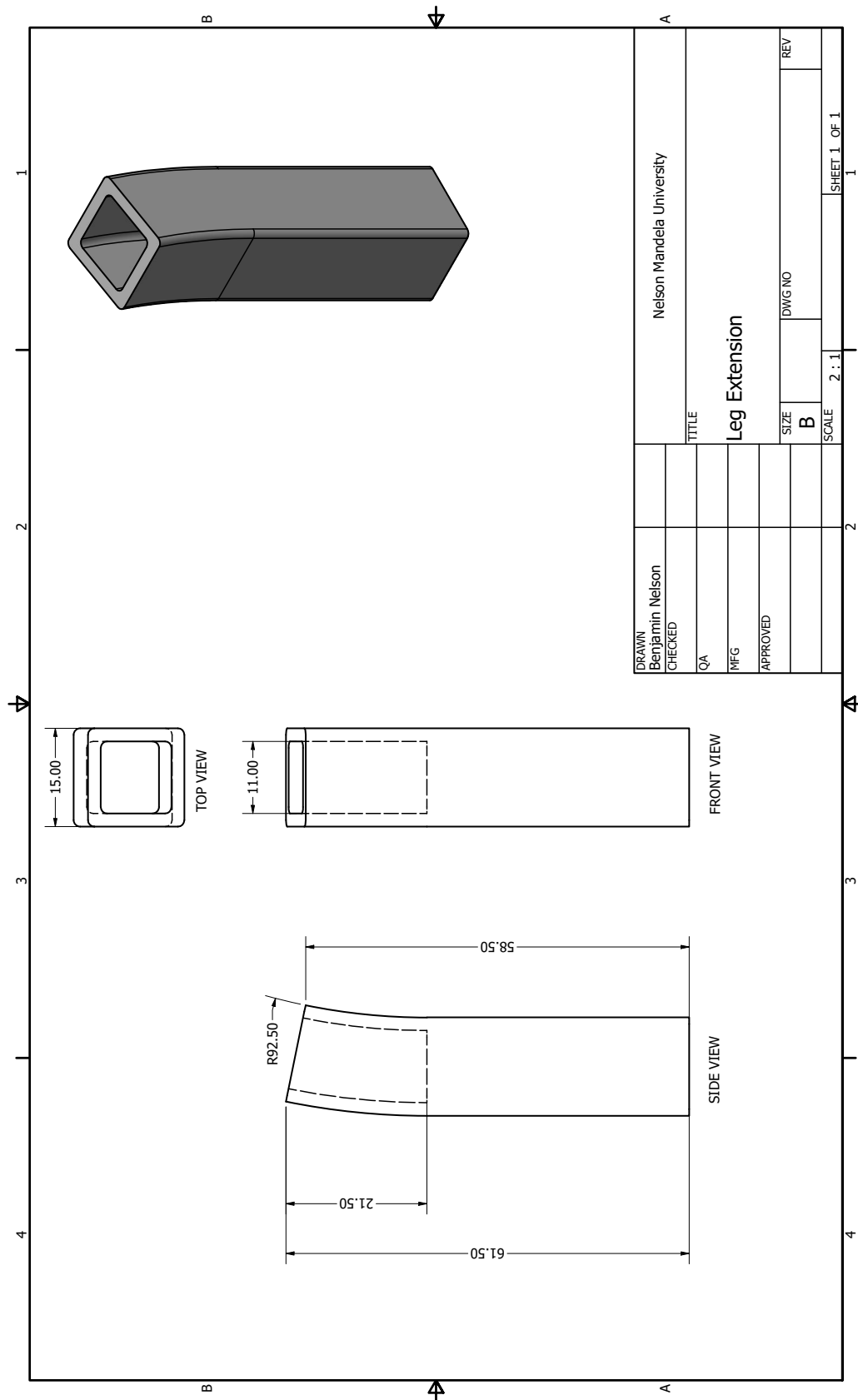
Appendix 3.1: Gimbal Attachment v1 CAD Drawing



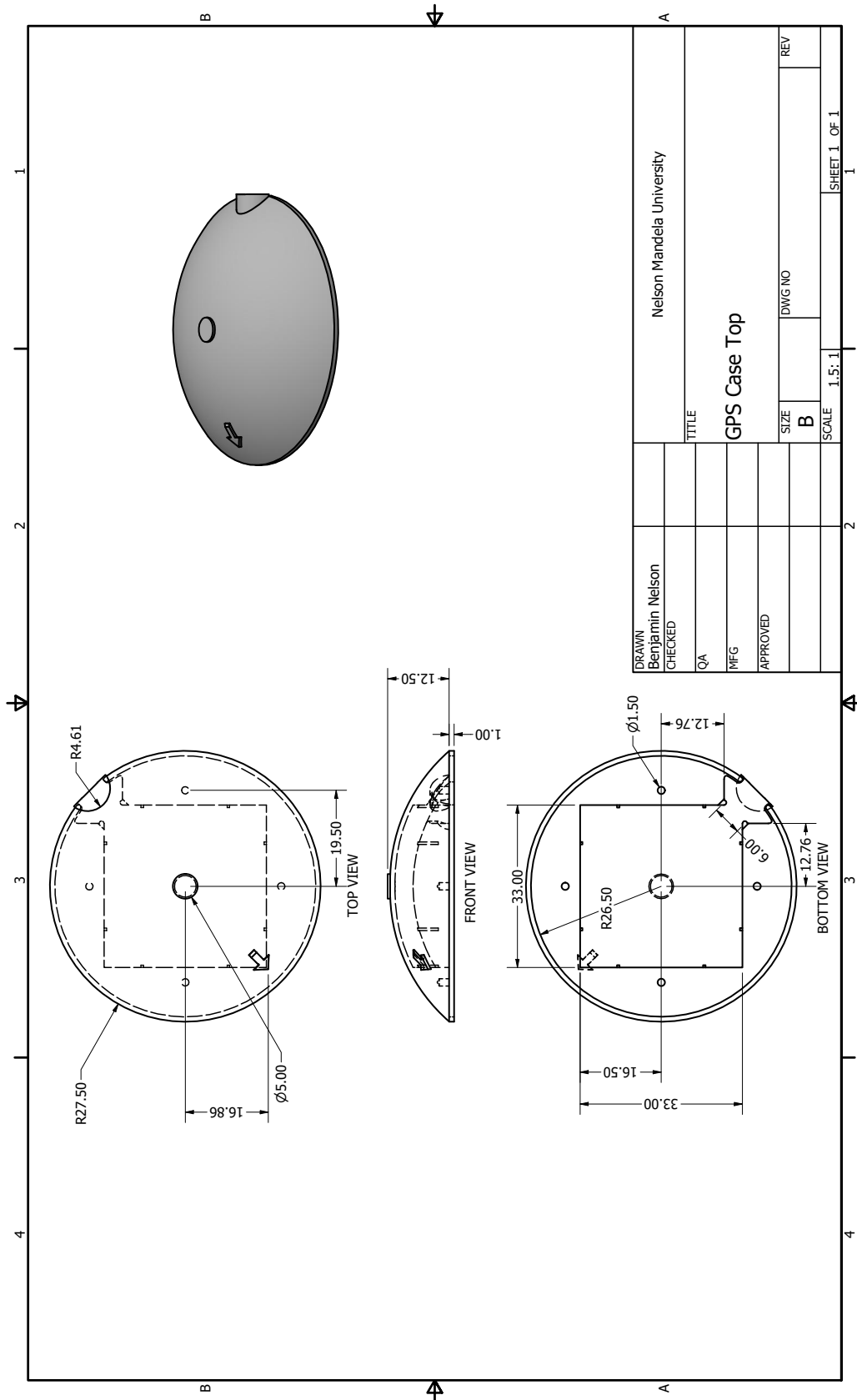
Appendix 3.2: Gimbal Attachment v2 CAD Drawing

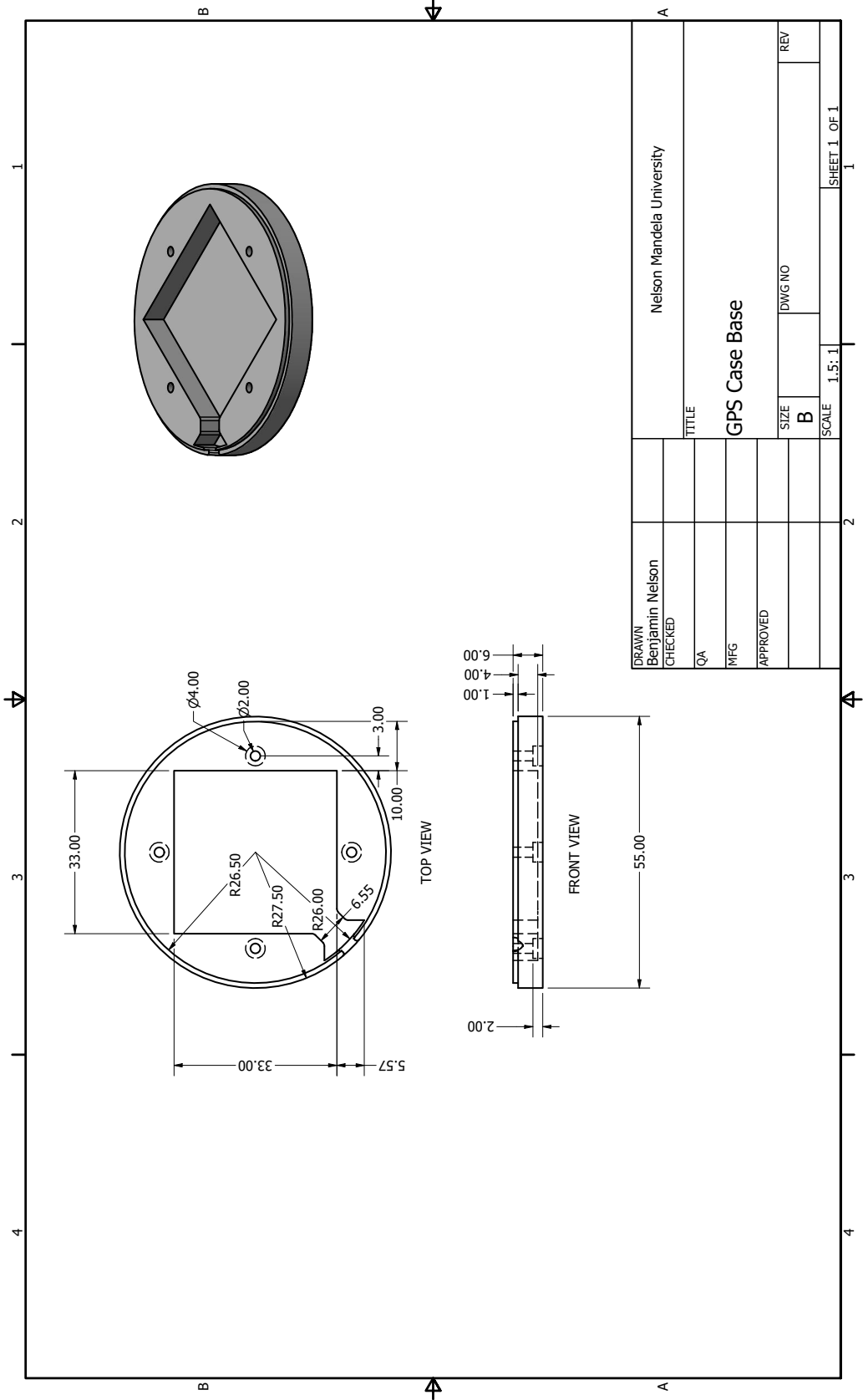


Appendix 3.3: Leg Extension CAD Drawing



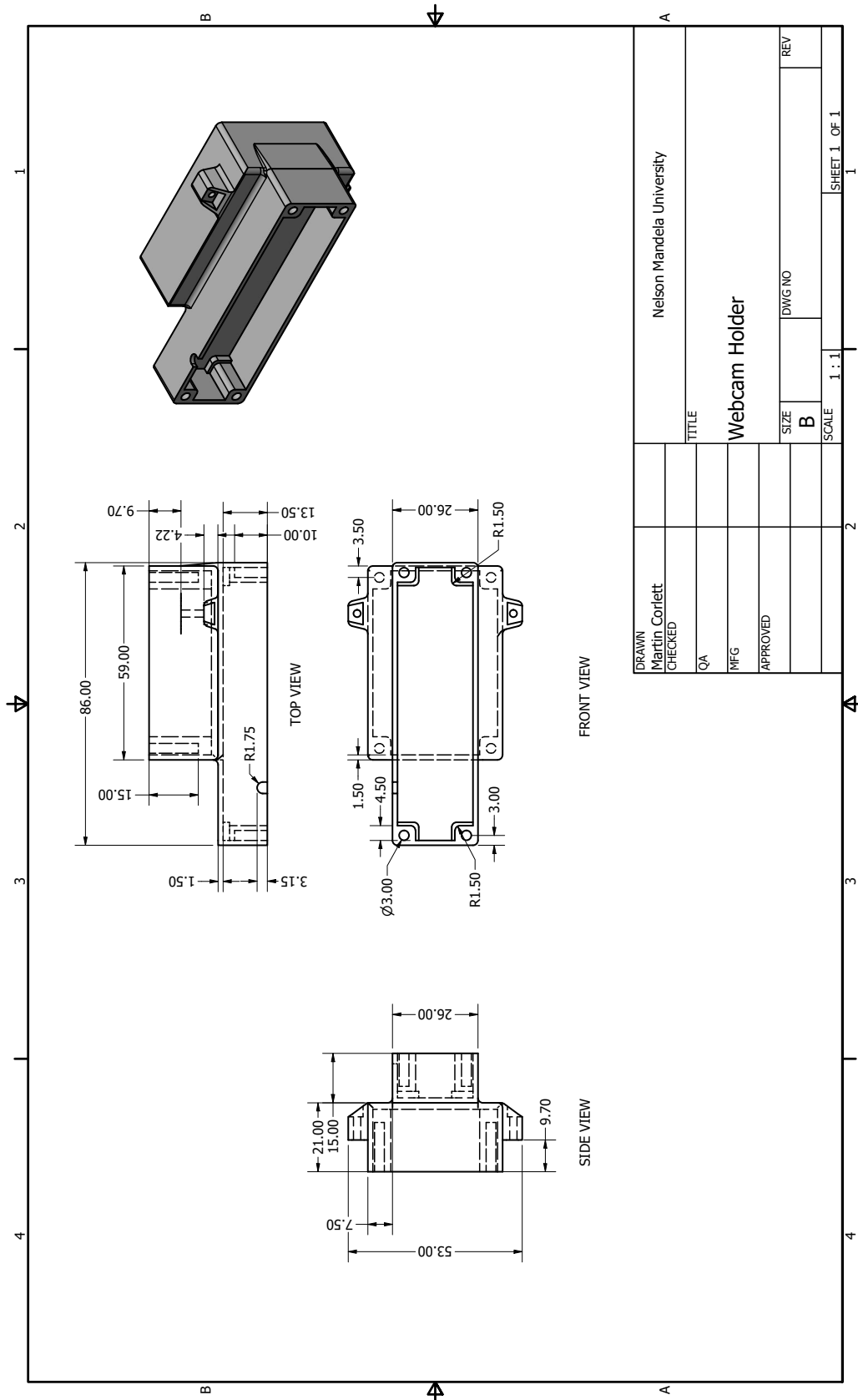
Appendix 3.4: GPS Case CAD Drawings



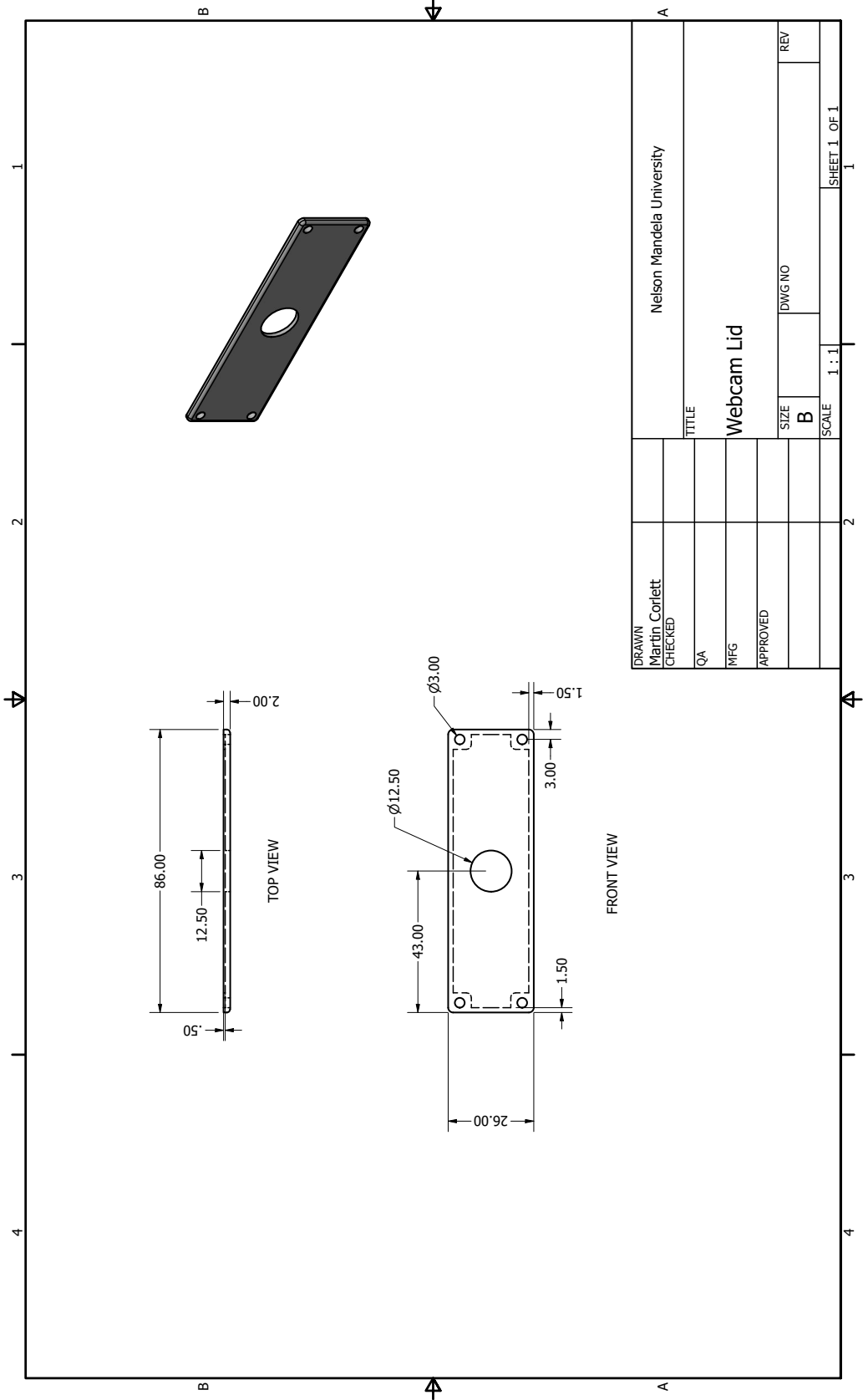


DRAWN Benjamin Nelson	Nelson Mandela University	
CHECKED	TITLE	
QA	GPS Case Base	
MFG	SIZE B	DWG NO
APPROVED	SCALE 1:5: 1	REV
		SHEET 1 OF 1

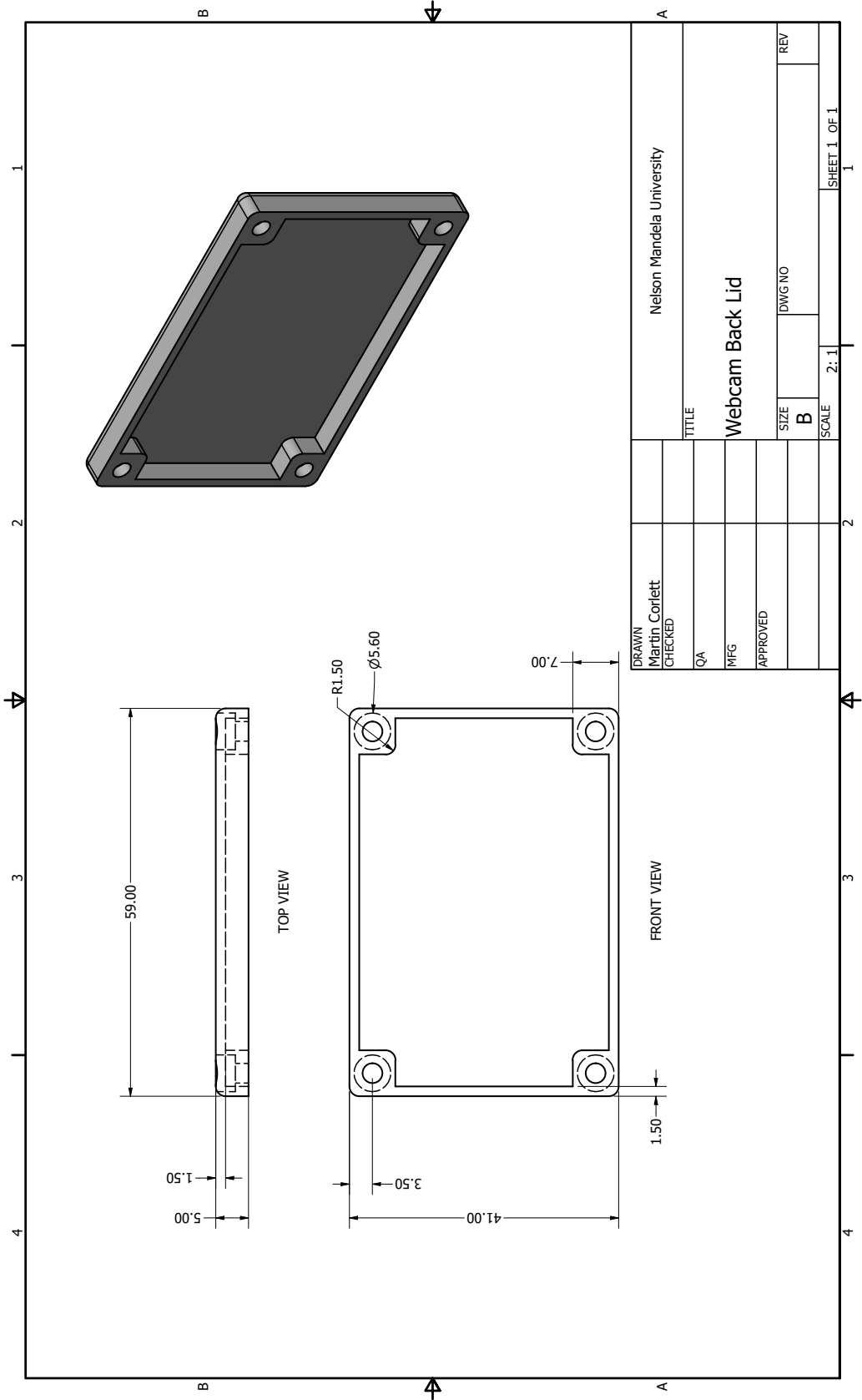
Appendix 3.5: Webcam CAD Drawings



DRAWN Martin Corlett		Nelson Mandela University	
CHECKED		TITLE	
QA		Webcam Holder	
MFG		SIZE	DWG NO
APPROVED		B	REV
		SCALE	SHEET 1 OF 1
		1:1	1

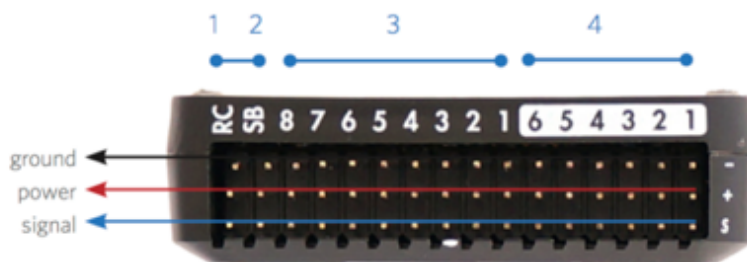
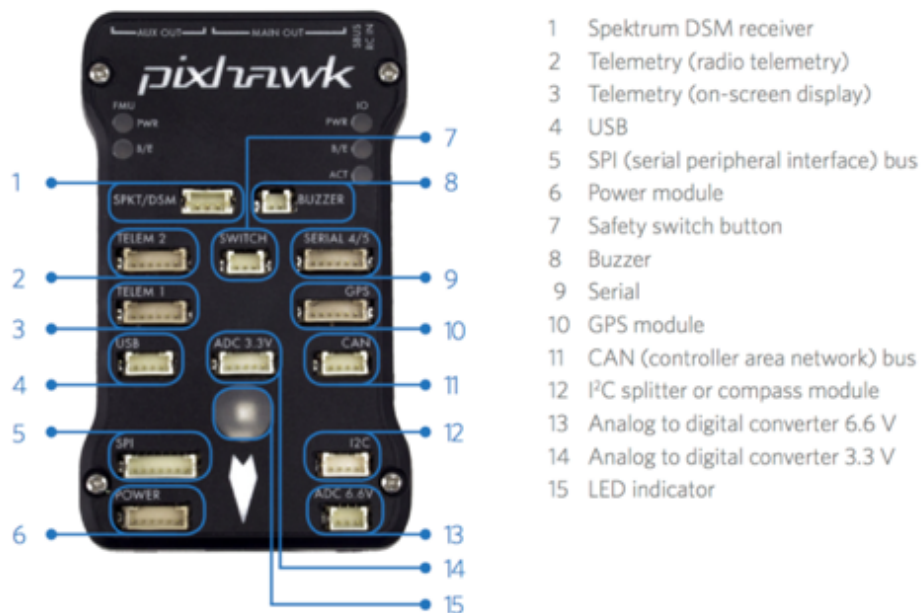


DRAWN	Martin Corlett				Nelson Mandela University
CHECKED					
QA					TITLE
MFG					Webcam Lid
APPROVED					SIZE
					B
					DWG NO
					REV
					SCALE
					1:1
					SHEET 1 OF 1



DRAWN Martin Corlett		Nelson Mandela University	
CHECKED		TITLE	
QA		Webcam Back Lid	
MFG		SIZE	DWG NO
APPROVED		B	REV
		SCALE	SHEET 1 OF 1
		2:1	1

Appendix 3.6: Pixhawk Schematic and Pinout



TELEM1 & TELEM2 Ports		
Pin	Signal	Volt
1 (red)	VCC	+5V
2 (blk)	TX (OUT)	+3.3V
3 (blk)	RX (IN)	+3.3V
4 (blk)	CTS (IN)	+3.3V
5 (blk)	RTS (OUT)	+3.3V
6 (blk)	GND	GND

GPS Port		
Pin	Signal	Volt
1 (red)	VCC	+5V
2 (blk)	TX (OUT)	+3.3V
3 (blk)	RX (IN)	+3.3V
4 (blk)	CAN2 TX	+3.3V
5 (blk)	CAN2 RX	+3.3V
6 (blk)	GND	GND

SERIAL 4/5 Port		
Pin	Signal	Volt
1 (red)	VCC	+5V
2 (blk)	TX (#4)	+3.3V
3 (blk)	RX (#4)	+3.3V
4 (blk)	TX (#5)	+3.3V
5 (blk)	RX (#5)	+3.3V
6 (blk)	GND	GND

ADC 6.6V		
Pin	Signal	Volt
1 (red)	VCC	+5V
2 (blk)	ADC IN	up to +6.6V
3 (blk)	GND	GND

ADC 3.3V		
Pin	Signal	Volt
1 (red)	VCC	+5V
2 (blk)	ADC IN	up to +3.3V
3 (blk)	GND	GND
4 (blk)	ADC IN	up to +3.3V
5 (blk)	GND	GND

I2C		
Pin	Signal	Volt
1 (red)	VCC	+5V
2 (blk)	SCL	+3.3 (pullups)
3 (blk)	SDA	+3.3 (pullups)
4 (blk)	GND	GND

CAN		
Pin	Signal	Volt
1 (red)	VCC	+5V
2 (blk)	CAN_H	+12V
3 (blk)	CAN_L	+12V
4 (blk)	GND	GND

SPI		
Pin	Signal	Volt
1 (red)	VCC	+5V
2 (blk)	SPI_EXT_SCK	+3.3V
3 (blk)	SPI_EXT_MISO	+3.3V
4 (blk)	SPI_EXT_MOSI	+3.3V
5 (blk)	!GPIO_EXT	+3.3V
6 (blk)	!SPI_EXT_NSS	+3.3V
7 (blk)	GND	GND

POWER		
Pin	Signal	Volt
1 (red)	VCC	+5V
2 (blk)	VCC	+5V
3 (blk)	CURRENT	+3.3V
4 (blk)	VOLTAGE	+3.3V
5 (blk)	GND	GND
6 (blk)	GND	GND

SWITCH		
Pin	Signal	Volt
1 (red)	VCC	+3.3V
2 (blk)	!IO_LED_SAFETY	GND
3 (blk)	SAFETY	GND

Appendix 3.7: Pixhawk PX4 Parameters

Parameter	Parameter Value	Parameter	Parameter Value
BAT_ADC_CHANNEL	-1	MPC_LAND_ALT2	5
BAT_A_PER_V	15.39103031	MPC_LAND_SPEED	0.600000024
BAT_CAPACITY	6600	MPC_MANTHR_MIN	0.079999998
BAT_CNT_V_CURR	0.000805664	MPC_MAN_TILT_MAX	35
BAT_CNT_V_VOLT	0.000805664	MPC_MAN_Y_MAX	200
BAT_CRIT_THR	0.07	MPC_POS_MODE	1
BAT_EMERGEN_THR	0.050000001	MPC_SPOOLUP_TIME	5
BAT_LOW_THR	0.150000006	MPC_THR_CURVE	0
BAT_N_CELLS	3	MPC_THR_HOVER	0.600000024
BAT_R_INTERNAL	-1	MPC_THR_MAX	1
BAT_SOURCE	0	MPC_THR_MIN	0.079999998
BAT_V_CHARGED	4.184000015	MPC_TILTMAX_AIR	45
BAT_V_DIV	10.17793941	MPC_TILTMAX_LND	12
BAT_V_EMPTY	3	MPC_TKO_RAMP_T	0.400000006
BAT_V_LOAD_DROP	0.5	MPC_TKO_SPEED	1.5
BAT_V_OFFS_CURR	0	MPC_VELD_LP	5
CAL_ACC0_EN	1	MPC_VEL_MANUAL	10
CAL_ACC0_ID	1246218	MPC_XY_CRUISE	5
CAL_ACC0_XOFF	-0.067638397	MPC_XY_MAN_EXPO	0
CAL_ACC0_XSCALE	0.991395772	MPC_XY_P	0.600000024
CAL_ACC0_YOFF	-0.072976589	MPC_XY_TRAJ_P	0.300000012
CAL_ACC0_YSCALE	1.005786061	MPC_XY_VEL_D	0.01
CAL_ACC0_ZOFF	-0.245393276	MPC_XY_VEL_I	0.02
CAL_ACC0_ZSCALE	0.994733572	MPC_XY_VEL_MAX	2
CAL_ACC1_EN	1	MPC_XY_VEL_P	0.090000004
CAL_ACC1_ID	1114634	MPC_YAWRAUTO_MAX	45
CAL_ACC1_XOFF	3.092870712	MPC_YAW_EXPO	0
CAL_ACC1_XSCALE	1.047754049	MPC_YAW_MODE	0
CAL_ACC1_YOFF	3.552666187	MPC_Z_MAN_EXPO	0
CAL_ACC1_YSCALE	1.000845194	MPC_Z_P	1
CAL_ACC1_ZOFF	2.505486012	MPC_Z_TRAJ_P	0.300000012
CAL_ACC1_ZSCALE	1.000477791	MPC_Z_VEL_D	0
CAL_ACC_PRIME	1246218	MPC_Z_VEL_I	0.02
CAL_AIR_CMODEL	0	MPC_Z_VEL_MAX_DN	0.5
CAL_AIR_TUBED_MM	1.5	MPC_Z_VEL_MAX_UP	3
CAL_AIR_TUBELEN	0.200000003	MPC_Z_VEL_P	0.200000003
CAL_BARO_PRIME	0	NAV_ACC_RAD	2
CAL_GYRO0_EN	1	NAV_AH_ALT	600

CAL_GYRO0_ID	2163722	NAV_AH_LAT	-265847810
CAL_GYRO0_XOFF	0.001806231	NAV_AH_LON	1518423250
CAL_GYRO0_XSCALE	1	NAV_DLL_ACT	0
CAL_GYRO0_YOFF	0.025406186	NAV_DLL_AH_T	120
CAL_GYRO0_YSCALE	1	NAV_DLL_CHSK	0
CAL_GYRO0_ZOFF	-0.018975759	NAV_DLL_CH_ALT	600
CAL_GYRO0_ZSCALE	1	NAV_DLL_CH_LAT	-266072120
CAL_GYRO1_EN	1	NAV_DLL_CH_LON	1518453890
CAL_GYRO1_ID	2228490	NAV_DLL_CH_T	120
CAL_GYRO1_XOFF	0.035897288	NAV_DLL_N	2
CAL_GYRO1_XSCALE	1	NAV_FORCE_VT	1
CAL_GYRO1_YOFF	0.03162903	NAV_FT_DST	8
CAL_GYRO1_YSCALE	1	NAV_FT_FS	1
CAL_GYRO1_ZOFF	0.029917274	NAV_FT_RS	0.5
CAL_GYRO1_ZSCALE	1	NAV_FW_ALTL_RAD	5
CAL_GYRO_PRIME	2163722	NAV_FW_ALT_RAD	10
CAL_MAG0_EN	1	NAV_GPSF_LT	0
CAL_MAG0_ID	73225	NAV_GPSF_P	0
CAL_MAG0_ROT	0	NAV_GPSF_R	15
CAL_MAG0_XOFF	0.080767058	NAV_GPSF_TR	0
CAL_MAG0_XSCALE	1.071887374	NAV_LOITER_RAD	50
CAL_MAG0_YOFF	0.015436961	NAV_MC_ALT_RAD	0.800000012
CAL_MAG0_YSCALE	0.892002106	NAV_MIN_FT_HT	8
CAL_MAG0_ZOFF	-0.107885525	NAV_RCL_ACT	2
CAL_MAG0_ZSCALE	1.059280038	NAV_RCL_LT	120
CAL_MAG1_EN	1	NAV_TRAFF_AVOID	1
CAL_MAG1_ID	131594	PLD_BTOUT	5
CAL_MAG1_ROT	-1	PLD_FAPPR_ALT	0.100000001
CAL_MAG1_XOFF	-0.157269895	PLD_HACC_RAD	0.200000003
CAL_MAG1_XSCALE	1.047619462	PLD_MAX_SRCH	3
CAL_MAG1_YOFF	0.27500093	PLD_SRCH_ALT	10
CAL_MAG1_YSCALE	0.966791272	PLD_SRCH_TOUT	10
CAL_MAG1_ZOFF	-0.138879791	PWM_AUX_DIS1	-1
CAL_MAG1_ZSCALE	0.998935401	PWM_AUX_DIS2	-1
CAL_MAG2_ID	0	PWM_AUX_DIS3	-1
CAL_MAG2_ROT	-1	PWM_AUX_DIS4	-1
CAL_MAG3_ID	0	PWM_AUX_DIS5	-1
CAL_MAG3_ROT	-1	PWM_AUX_DIS6	-1
CAL_MAG_PRIME	73225	PWM_AUX_DIS7	-1
CAL_MAG_SIDES	63	PWM_AUX_DIS8	-1
CBRK_AIRSPD_CHK	0	PWM_AUX_DISARMED	1500
CBRK_BUZZER	0	PWM_AUX_FAIL1	-1

CBRK_ENGINEFAIL	284953	PWM_AUX_FAIL2	-1
CBRK_FLIGHTTERM	121212	PWM_AUX_FAIL3	-1
CBRK_GPSFAIL	0	PWM_AUX_FAIL4	-1
CBRK_IO_SAFETY	22027	PWM_AUX_FAIL5	-1
CBRK_RATE_CTRL	0	PWM_AUX_FAIL6	-1
CBRK_SUPPLY_CHK	0	PWM_AUX_FAIL7	-1
CBRK_USB_CHK	0	PWM_AUX_FAIL8	-1
CBRK_VELPOSERR	0	PWM_AUX_MAX	2000
COM_ARM_AUTH	256010	PWM_AUX_MAX1	-1
COM_ARM_IMU_ACC	0.699999988	PWM_AUX_MAX2	-1
COM_ARM_IMU_GYR	0.25	PWM_AUX_MAX3	-1
COM_ARM_MAG	0.150000006	PWM_AUX_MAX4	-1
COM_ARM_MIS_REQ	0	PWM_AUX_MAX5	-1
COM_ARM_SWISBTN	0	PWM_AUX_MAX6	-1
COM_ARM_WO_GPS	1	PWM_AUX_MAX7	-1
COM_ASPD_FS_ACT	0	PWM_AUX_MAX8	-1
COM_ASPD_FS_DLY	0	PWM_AUX_MIN	1000
COM_ASPD_STALL	10	PWM_AUX_MIN1	-1
COM_DISARM_LAND	0.100000001	PWM_AUX_MIN2	-1
COM_DL_LOSS_T	10	PWM_AUX_MIN3	-1
COM_EF_C2T	5	PWM_AUX_MIN4	-1
COM_EF_THROT	0.5	PWM_AUX_MIN5	-1
COM_EF_TIME	10	PWM_AUX_MIN6	-1
COM_FLIGHT_UUID	108	PWM_AUX_MIN7	-1
COM_FLTMODE1	6	PWM_AUX_MIN8	-1
COM_FLTMODE2	8	PWM_AUX_REV1	0
COM_FLTMODE3	2	PWM_AUX_REV2	0
COM_FLTMODE4	3	PWM_AUX_REV3	0
COM_FLTMODE5	11	PWM_AUX_REV4	0
COM_FLTMODE6	5	PWM_AUX_REV5	0
COM_HLDL_LOSS_T	120	PWM_AUX_REV6	0
COM_HLDL_REG_T	0	PWM_AUX_TRIM1	0
COM_HOME_H_T	5	PWM_AUX_TRIM2	0
COM_HOME_V_T	10	PWM_AUX_TRIM3	0
COM_LOW_BAT_ACT	0	PWM_AUX_TRIM4	0
COM_OA_BOOT_T	100	PWM_AUX_TRIM5	0
COM_OBL_ACT	0	PWM_AUX_TRIM6	0
COM_OBL_RC_ACT	0	PWM_DISARMED	900
COM_OBS_AVOID	0	PWM_MAIN_DIS1	-1
COM_OF_LOSS_T	0	PWM_MAIN_DIS2	-1
COM_POSCTL_NAVL	0	PWM_MAIN_DIS3	-1
COM_POS_FS_DELAY	1	PWM_MAIN_DIS4	-1

COM_POS_FS_EPH	5	PWM_MAIN_DIS5	-1
COM_POS_FS_EPV	10	PWM_MAIN_DIS6	-1
COM_POS_FS_GAIN	10	PWM_MAIN_DIS7	-1
COM_POS_FS_PROB	30	PWM_MAIN_DIS8	-1
COM_RC_ARM_HYST	1000	PWM_MAIN_FAIL1	-1
COM_RC_IN_MODE	0	PWM_MAIN_FAIL2	-1
COM_RC_LOSS_T	0.5	PWM_MAIN_FAIL3	-1
COM_RC_OVERRIDE	0	PWM_MAIN_FAIL4	-1
COM_RC_STICK_OV	12	PWM_MAIN_FAIL5	-1
COM_TAKEOFF_ACT	0	PWM_MAIN_FAIL6	-1
COM_TAS_FS_INNOV	1	PWM_MAIN_FAIL7	-1
COM_TAS_FS_INTEG	-1	PWM_MAIN_FAIL8	-1
COM_TAS_FS_T1	3	PWM_MAIN_MAX1	-1
COM_TAS_FS_T2	100	PWM_MAIN_MAX2	-1
COM_VEL_FS_EVH	1	PWM_MAIN_MAX3	-1
EKF2_ABIAS_INIT	0.200000003	PWM_MAIN_MAX4	-1
EKF2_ABL_ACCLIM	25	PWM_MAIN_MAX5	-1
EKF2_ABL_GYRLIM	3	PWM_MAIN_MAX6	-1
EKF2_ABL_LIM	0.400000006	PWM_MAIN_MAX7	-1
EKF2_ABL_TAU	0.5	PWM_MAIN_MAX8	-1
EKF2_ACC_B_NOISE	0.003	PWM_MAIN_MIN1	-1
EKF2_ACC_NOISE	0.349999994	PWM_MAIN_MIN2	-1
EKF2_AID_MASK	1	PWM_MAIN_MIN3	-1
EKF2_ANGERR_INIT	0.100000001	PWM_MAIN_MIN4	-1
EKF2_ARSP_THR	0	PWM_MAIN_MIN5	-1
EKF2_ASPD_MAX	20	PWM_MAIN_MIN6	-1
EKF2_ASP_DELAY	100	PWM_MAIN_MIN7	-1
EKF2_AVEL_DELAY	5	PWM_MAIN_MIN8	-1
EKF2_BARO_DELAY	0	PWM_MAIN_REV1	0
EKF2_BARO_GATE	5	PWM_MAIN_REV2	0
EKF2_BARO_NOISE	2	PWM_MAIN_REV3	0
EKF2_BCOEF_X	25	PWM_MAIN_REV4	0
EKF2_BCOEF_Y	25	PWM_MAIN_REV5	0
EKF2_BETA_GATE	5	PWM_MAIN_REV6	0
EKF2_BETA_NOISE	0.300000012	PWM_MAIN_REV7	0
EKF2_DECL_TYPE	7	PWM_MAIN_REV8	0
EKF2_DRAG_NOISE	2.5	PWM_MAIN_TRIM1	0
EKF2_EAS_NOISE	1.399999976	PWM_MAIN_TRIM2	0
EKF2_EVA_NOISE	0.050000001	PWM_MAIN_TRIM3	0
EKF2_EVP_NOISE	0.050000001	PWM_MAIN_TRIM4	0
EKF2_EV_DELAY	175	PWM_MAIN_TRIM5	0
EKF2_EV_GATE	5	PWM_MAIN_TRIM6	0

EKF2_EV_POS_X	0	PWM_MAIN_TRIM7	0
EKF2_EV_POS_Y	0	PWM_MAIN_TRIM8	0
EKF2_EV_POS_Z	0	PWM_MAX	1950
EKF2_FUSE_BETA	0	PWM_MIN	1075
EKF2_GBIAS_INIT	0.100000001	PWM_RATE	400
EKF2_GND_EFF_DZ	0	PWM_SBUS_MODE	0
EKF2_GND_MAX_HGT	0.5	RC10_DZ	0
EKF2_GPS_CHECK	245	RC10_MAX	2000
EKF2_GPS_DELAY	110	RC10_MIN	1000
EKF2_GPS_MASK	0	RC10_REV	1
EKF2_GPS_POS_X	0	RC10_TRIM	1500
EKF2_GPS_POS_Y	0	RC11_DZ	0
EKF2_GPS_POS_Z	0	RC11_MAX	2000
EKF2_GPS_P_GATE	5	RC11_MIN	1000
EKF2_GPS_P_NOISE	0.5	RC11_REV	1
EKF2_GPS_TAU	10	RC11_TRIM	1500
EKF2_GPS_V_GATE	5	RC12_DZ	0
EKF2_GPS_V_NOISE	0.5	RC12_MAX	2000
EKF2_GYR_B_NOISE	0.001	RC12_MIN	1000
EKF2_GYR_NOISE	0.015	RC12_REV	1
EKF2_HDG_GATE	2.599999905	RC12_TRIM	1500
EKF2_HEAD_NOISE	0.300000012	RC13_DZ	0
EKF2_HGT_MODE	0	RC13_MAX	2000
EKF2_IMU_POS_X	0	RC13_MIN	1000
EKF2_IMU_POS_Y	0	RC13_REV	1
EKF2_IMU_POS_Z	0	RC13_TRIM	1500
EKF2_MAGBIAS_ID	73225	RC14_DZ	0
EKF2_MAGBIAS_X	0.001802281	RC14_MAX	2000
EKF2_MAGBIAS_Y	-0.001675414	RC14_MIN	1000
EKF2_MAGBIAS_Z	-4.92853E-05	RC14_REV	1
EKF2_MAGB_K	0.200000003	RC14_TRIM	1500
EKF2_MAGB_VREF	2.5E-07	RC15_DZ	0
EKF2_MAG_ACCLIM	0.5	RC15_MAX	2000
EKF2_MAG_B_NOISE	1E-04	RC15_MIN	1000
EKF2_MAG_DECL	-23.14025879	RC15_REV	1
EKF2_MAG_DELAY	0	RC15_TRIM	1500
EKF2_MAG_E_NOISE	0.001	RC16_DZ	0
EKF2_MAG_GATE	3	RC16_MAX	2000
EKF2_MAG_NOISE	0.050000001	RC16_MIN	1000
EKF2_MAG_TYPE	0	RC16_REV	1
EKF2_MAG_YAWLIM	0.25	RC16_TRIM	1500
EKF2_MIN_OBS_DT	20	RC17_DZ	0

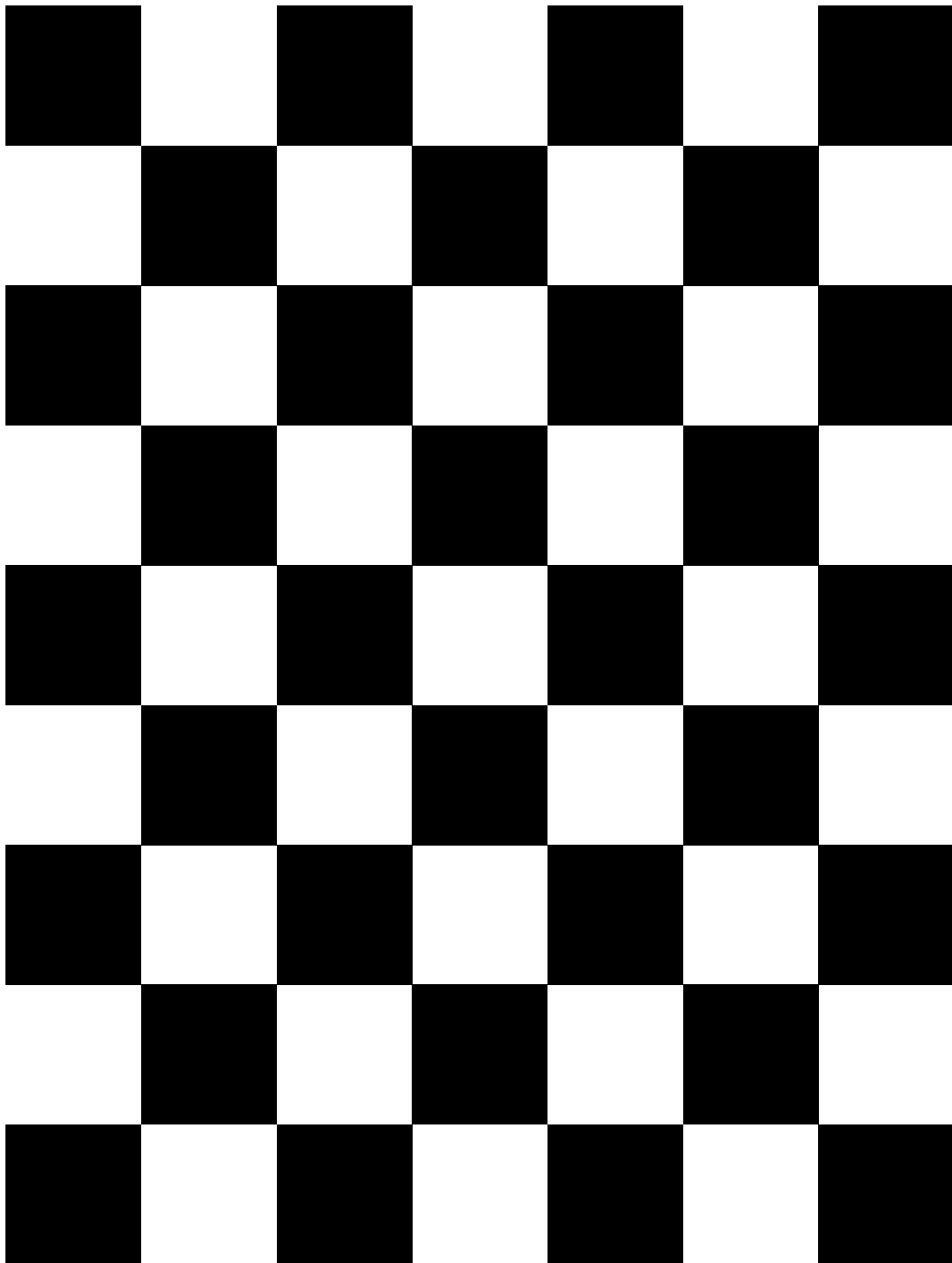
EKF2_MIN_RNG	0.100000001	RC17_MAX	2000
EKF2_MOVE_TEST	1	RC17_MIN	1000
EKF2_NOAID_NOISE	10	RC17_REV	1
EKF2_NOAID_TOUT	5000000	RC17_TRIM	1500
EKF2_OF_DELAY	5	RC18_DZ	0
EKF2_OF_GATE	3	RC18_MAX	2000
EKF2_OF_N_MAX	0.5	RC18_MIN	1000
EKF2_OF_N_MIN	0.150000006	RC18_REV	1
EKF2_OF_POS_X	0	RC18_TRIM	1500
EKF2_OF_POS_Y	0	RC1_DZ	10
EKF2_OF_POS_Z	0	RC1_MAX	2003
EKF2_OF_QMIN	1	RC1_MIN	982
EKF2_PCOEF_XN	0	RC1_REV	1
EKF2_PCOEF_XP	0	RC1_TRIM	982
EKF2_PCOEF_YN	0	RC2_DZ	10
EKF2_PCOEF_YP	0	RC2_MAX	2006
EKF2_PCOEF_Z	0	RC2_MIN	982
EKF2_REQ_EPH	3	RC2_REV	1
EKF2_REQ_EPV	5	RC2_TRIM	1500
EKF2_REQ_GDOP	2.5	RC3_DZ	10
EKF2_REQ_HDRIFT	0.100000001	RC3_MAX	2006
EKF2_REQ_NSATS	6	RC3_MIN	982
EKF2_REQ_SACC	0.5	RC3_REV	1
EKF2_REQ_VDRIFT	0.200000003	RC3_TRIM	1498
EKF2_RNG_AID	0	RC4_DZ	10
EKF2_RNG_A_HMAX	5	RC4_MAX	2006
EKF2_RNG_A_IGATE	1	RC4_MIN	982
EKF2_RNG_A_VMAX	1	RC4_REV	1
EKF2_RNG_DELAY	5	RC4_TRIM	1501
EKF2_RNG_GATE	5	RC5_DZ	10
EKF2_RNG_NOISE	0.100000001	RC5_MAX	2006
EKF2_RNG_PITCH	0	RC5_MIN	982
EKF2_RNG_POS_X	0	RC5_REV	1
EKF2_RNG_POS_Y	0	RC5_TRIM	1494
EKF2_RNG_POS_Z	0	RC6_DZ	10
EKF2_RNG_SFE	0.050000001	RC6_MAX	2006
EKF2_TAS_GATE	3	RC6_MIN	982
EKF2_TAU_POS	0.25	RC6_REV	1
EKF2_TAU_VEL	0.25	RC6_TRIM	1494
EKF2_TERR_GRAD	0.5	RC7_DZ	10
EKF2_TERR_NOISE	5	RC7_MAX	2006
EKF2_WIND_NOISE	0.100000001	RC7_MIN	982

EV_TSK_RC_LOSS	0	RC7_REV	1
EV_TSK_STAT_DIS	0	RC7_TRIM	1494
FD_FAIL_P	60	RC8_DZ	10
FD_FAIL_R	60	RC8_MAX	2006
FW_MAN_P_SC	1	RC8_MIN	982
FW_MAN_R_SC	1	RC8_REV	1
FW_MAN_Y_SC	1	RC8_TRIM	1494
GF_ACTION	1	RC9_DZ	0
GF_ALTMODE	0	RC9_MAX	2006
GF_COUNT	-1	RC9_MIN	982
GF_MAX_HOR_DIST	0	RC9_REV	1
GF_MAX_VER_DIST	0	RC9_TRIM	1494
GF_SOURCE	0	RC_ACRO_TH	0.5
GPS_1_CONFIG	201	RC_ARMSWITCH_TH	0.25
GPS_2_CONFIG	0	RC_ASSIST_TH	0.25
GPS_DUMP_COMM	0	RC_AUTO_TH	0.75
GPS_UBX_DYNMODEL	7	RC_CHAN_CNT	18
GPS_YAW_OFFSET	0	RC_FAILS_THR	0
IMU_ACCEL_CUTOFF	30	RC_FLT_CUTOFF	10
IMU_GYRO_CUTOFF	30	RC_FLT_SMP_RATE	50
LED_RGB_MAXBRT	15	RC_GEAR_TH	0.25
LNDMC_ALT_MAX	-1	RC_KILLSWITCH_TH	0.25
LNDMC_FFALL_THR	2	RC_LOITER_TH	0.5
LNDMC_FFALL_TTRI	0.300000012	RC_MAN_TH	0.5
LNDMC_LOW_T_THR	0.300000012	RC_MAP_ACRO_SW	0
LNDMC_ROT_MAX	20	RC_MAP_ARM_SW	6
LNDMC_XY_VEL_MAX	1.5	RC_MAP_AUX1	7
LNDMC_Z_VEL_MAX	0.5	RC_MAP_AUX2	8
LND_FLIGHT_T_HI	0	RC_MAP_AUX3	0
LND_FLIGHT_T_LO	-509819056	RC_MAP_AUX4	0
MAV_0_CONFIG	101	RC_MAP_AUX5	0
MAV_0_FORWARD	1	RC_MAP_AUX6	0
MAV_0_MODE	0	RC_MAP_FAILSAFE	0
MAV_0_RATE	1200	RC_MAP_FLAPS	0
MAV_1_CONFIG	102	RC_MAP_FLTMODE	5
MAV_1_FORWARD	1	RC_MAP_GEAR_SW	0
MAV_1_MODE	2	RC_MAP_KILL_SW	0
MAV_1_RATE	0	RC_MAP_LOITER_SW	0
MAV_2_CONFIG	0	RC_MAP_MAN_SW	0
MAV_BROADCAST	0	RC_MAP_MODE_SW	0
MAV_COMP_ID	1	RC_MAP_OFFB_SW	9
MAV_FWDEXTSP	1	RC_MAP_PARAM1	0

MAV_HASH_CHK_EN	1	RC_MAP_PARAM2	0
MAV_HB_FORW_EN	1	RC_MAP_PARAM3	0
MAV_ODOM_LP	0	RC_MAP_PITCH	3
MAV_PROTO_VER	0	RC_MAP_POSCTL_SW	0
MAV_RADIO_ID	0	RC_MAP_RATT_SW	0
MAV_SYS_ID	1	RC_MAP_RETURN_SW	0
MAV_TYPE	13	RC_MAP_ROLL	2
MAV_USEHILGPS	0	RC_MAP_STAB_SW	0
MC_ACRO_EXPO	0.689999998	RC_MAP_THROTTLE	1
MC_ACRO_EXPO_Y	0.689999998	RC_MAP_TRANS_SW	0
MC_ACRO_P_MAX	720	RC_MAP_YAW	4
MC_ACRO_R_MAX	720	RC_OFFB_TH	0.5
MC_ACRO_SUPEXPO	0.699999988	RC_POSCTL_TH	0.5
MC_ACRO_SUPEXPOY	0.699999988	RC_RATT_TH	0.5
MC_ACRO_Y_MAX	540	RC_RETURN_TH	0.5
MC_AIRMODE	0	RC_RSSI_PWM_CHAN	0
MC_BAT_SCALE_EN	1	RC_RSSI_PWM_MAX	1000
MC_DTERM_CUTOFF	0	RC_RSSI_PWM_MIN	2000
MC_PITCHRATE_D	0.003	RC_STAB_TH	0.5
MC_PITCHRATE_FF	0	RC_TRANS_TH	0.25
MC_PITCHRATE_I	0.200000003	RTL_DESCEND_ALT	5
MC_PITCHRATE_MAX	220	RTL_LAND_DELAY	0
MC_PITCHRATE_P	0.150000006	RTL_MIN_DIST	20
MC_PITCH_P	6.5	RTL_RETURN_ALT	5
MC_PR_INT_LIM	0.300000012	RTL_TYPE	0
MC_RATT_TH	0.800000012	SDLOG_DIRS_MAX	0
MC_ROLLRATE_D	0.003	SDLOG_MISSION	0
MC_ROLLRATE_FF	0	SDLOG_MODE	0
MC_ROLLRATE_I	0.200000003	SDLOG_PROFILE	3
MC_ROLLRATE_MAX	220	SDLOG_UTC_OFFSET	0
MC_ROLLRATE_P	0.150000006	SENS_BARO_QNH	1013.25
MC_ROLL_P	6.5	SENS_BOARD_ROT	0
MC_RR_INT_LIM	0.300000012	SENS_BOARD_X_OFF	0.021766607
MC_TPA_BREAK_D	1	SENS_BOARD_Y_OFF	1.469085574
MC_TPA_BREAK_I	1	SENS_BOARD_Z_OFF	0
MC_TPA_BREAK_P	1	SENS_DPRES_ANSC	0
MC_TPA_RATE_D	0	SENS_DPRES_OFF	0
MC_TPA_RATE_I	0	SENS_EN_LL40LS	0
MC_TPA_RATE_P	0	SENS_EN_THERMAL	-1
MC_YAWRATE_D	0	SENS_FLOW_MAXHGT	3
MC_YAWRATE_FF	0	SENS_FLOW_MAXR	2.5
MC_YAWRATE_I	0.100000001	SENS_FLOW_MINHGT	0.699999988

MC_YAWRATE_MAX	125	SENS_FLOW_ROT	6
MC_YAWRATE_P	0.200000003	SER_GPS1_BAUD	0
MC_YAW_P	2.799999952	SER_TEL1_BAUD	57600
MC_YR_INT_LIM	0.300000012	SER_TEL2_BAUD	921600
MIS_ALTMODE	1	SYS_AUTOCONFIG	0
MIS_DIST_1WP	900	SYS_AUTOSTART	6001
MIS_DIST_WPS	900	SYS_BL_UPDATE	0
MIS_LTRMIN_ALT	-1	SYS_CAL_ACCEL	0
MIS_MNT_YAW_CTL	0	SYS_CAL_BARO	0
MIS_TAKEOFF_ALT	2.5	SYS_CAL_GYRO	0
MIS_TAKEOFF_REQ	0	SYS_CAL_TDEL	24
MIS_YAW_ERR	12	SYS_CAL_TMAX	10
MIS_YAW_TMT	-1	SYS_CAL_TMIN	5
MNT_MODE_IN	-1	SYS_COMPANION	0
MOT_ORDERING	0	SYS_FMU_TASK	0
MOT_SLEW_MAX	0	SYS_HAS_BARO	1
MPC_ACC_DOWN_MAX	10	SYS_HAS_MAG	1
MPC_ACC_HOR	5	SYS_HITL	0
MPC_ACC_HOR_ESTM	0.5	SYS_MC_EST_GROUP	2
MPC_ACC_HOR_MAX	10	SYS_PARAM_VER	1
MPC_ACC_UP_MAX	10	SYS_RESTART_TYPE	0
MPC_ALT_MODE	0	SYS_STCK_EN	1
MPC_AUTO_MODE	1	SYS_USE_IO	1
MPC_COL_PREV_D	-1	TC_A_ENABLE	0
MPC_CRUISE_90	3	TC_B_ENABLE	0
MPC_DEC_HOR_SLOW	5	TC_G_ENABLE	0
MPC_HOLD_DZ	0.100000001	THR_MDL_FAC	0
MPC_HOLD_MAX_XY	0.800000012	TRIM_PITCH	0
MPC_HOLD_MAX_Z	0.600000024	TRIM_ROLL	0
MPC_JERK_AUTO	8	TRIM_YAW	0
MPC_JERK_MAX	20	VT_B_DEC_MSS	2
MPC_JERK_MIN	8	VT_B_REV_DEL	0
MPC_LAND_ALT1	10		

Appendix 4.1: Checkerboard Used for Camera Calibration



Appendix 5.1: determinepose.cpp node

```
//relevant libraries used in this node
#include <ros/ros.h>
#include <geometry_msgs/PoseStamped.h>
#include <Eigen/Core>
#include <mavros_msgs/CommandBool.h>
#include <mavros_msgs/SetMode.h>
#include <mavros_msgs/State.h>
#include <std_msgs/String.h>
#include <std_msgs/Float64MultiArray.h>
#include <std_msgs/MultiArrayLayout.h>
#include <std_msgs/MultiArrayDimension.h>
#include <std_msgs/Float64.h>
#include <tf/tf.h>
#include <string>
#include <sstream>
#include <Eigen/Geometry>
#include <sensor_msgs/Imu.h>
#include <math.h>
#include <mavros_msgs/ParamValue.h>
#include <mavros_msgs/ParamSet.h>
#include <mavros_msgs/ParamPush.h>
#include <iostream>

// this allows identifiers to be used
using namespace std;
using namespace Eigen;

// declaring variables
geometry_msgs::PoseStamped final_pose;
geometry_msgs::PoseStamped local_pose;
mavros_msgs::State current_state;
mavros_msgs::ParamValue ValueOfParameter;
mavros_msgs::ParamSet SetParameter;
mavros_msgs::ParamPush PushParameter;

Quaternionf LocalPoseQuat;
Quaternionf IMUQuat;
Quaternionf ArucoQuat;
Quaternionf FinalRotation;
Quaternionf NewArucoQuat;
Quaternionf GeneralQuat;

std_msgs::String CoordinatesMsg;
std_msgs::String VelocityMsg;
std_msgs::String DescentMsg;
std_msgs::String AutolandMsg;
```



```

std_msgs::String FlightmodeMsg;
std_msgs::String YawMsg;
std_msgs::String QuatMsg;
std_msgs::String MultipleCoordinatesMsg;

std_msgs::Float64MultiArray DeterminedAircraftPoseMsg; // {orientation w, orientation x,
orientation y, orientation z, position x, position y, position z}

std_msgs::String StatusMsg;
string CoordinatesTemp;
string VelocityTemp;
string AutolandTemp;
string DescentTemp;
string FlightmodeTemp;
string YawTemp;
string QuatTemp;
string MultipleCoordinatesTemp;

string sAxis = "z";
string sPos = "2";
string sVelocity = "1";
string sAutoland = "false";
string sDescent = "0.2";
string sFlightmode = "";
string sYaw = "0";
string sQuatX = "0";
string sQuatY = "0";
string sQuatZ = "0";
string sQuatW = "0";
string sFollowX = "0";
string sFollowY = "0";

float fVelocity = 1.5;
float fDescent = 0.2;
float fPos = 2;
float AverageHomeZ = 0;
//float CamRatioValue = 4.0;
float CamRatioValue = 0.2;
float HeightWithRatio = 0;
float MarkerHeight = 0;
float fYawDegrees = 0;
float fYawRadians = 0;
float fQuatX = 0;
float fQuatY = 0;
float fQuatZ = 0;
float fQuatW = 0;
float ArucoOrientation[4] = {0,0,0,0};
float MaintainHeight = 0;
float MaintainX = 0;
float MaintainY = 0;

```

```

float YawEuler = 0;
float fFollowX = 0;
float fFollowY = 0;
float fFollowZ = 10;
float RelativeAltitude = 0;

bool bVelCallback = false;
bool bAutoMove = false;
bool SetParameterSuccess;
bool PushParameterSuccess;
bool bTakeoff = false;
bool bArmed = false;
bool bOffboard = false;
bool bStabilize = false;
bool bStatus = false;
bool bRotate = false;
bool bFirstRotation = false;
bool bFirstSetpoint = false;
bool bManual = true;
bool bHover = false;
bool bStartTimer = false;
bool bStartRotation = false;
bool bBusyRotating = false;
bool bAllowedToLand = false;
bool bFollow = false;
bool bCoordinatesReceived = false;
bool DisplayedAlready = false;
bool bActivateDisarm = false;
bool bLandAlreadyActivated = false;

ros::Time rotate_time;
ros::Time follow_time;

int iCount = 0;
int AverageHomeCount = 0;

// declaring a 3x3 matrix
Matrix3f R;
Matrix3f Y;

// Callback for the state of the aircraft
void state_cb(const mavros_msgs::State::ConstPtr& msg)
{
    current_state = *msg;
}

// vectors to store position before and after
Vector3f positionbe;
Vector3f positionaf;

```

```

//Change the horizontal velocity of the aircraft by making use of ParamSet and ParamPush
void ChangeVelocity()
{
    ros::NodeHandle n;
    ros::Rate r(20.0);

    ros::ServiceClient param_set_client = n.serviceClient<mavros_msgs::ParamSet>
        ("mavros/param/set");
    ros::ServiceClient param_push_client = n.serviceClient<mavros_msgs::ParamPush>
        ("mavros/param/push");

    ValueOfParameter.real = fVelocity;
    ValueOfParameter.integer = 0;
    SetParameter.request.value = ValueOfParameter;
    SetParameter.request.param_id = "MPC_XY_VEL_MAX";

    SetParameterSuccess = param_set_client.call(SetParameter);
    PushParameterSuccess = param_push_client.call(PushParameter);

    if( SetParameterSuccess && PushParameterSuccess )
    {
        printf("Velocity SUCCESSFULLY UPDATED with: %.1f \n", fVelocity);
        printf("\n");
        StatusMsg.data = "Velocity SUCCESSFULLY UPDATED";
    }
    bStatus = true;
    //    bVelCallback = false;
}
else
{
    printf("Velocity UPDATE FAILED \n");
    printf("\n");
    StatusMsg.data = "Velocity UPDATE FAILED";
}
bStatus = true;
}

    ros::spinOnce();
    r.sleep();
}

```

```

//Change the descending velocity of the aircraft by making use of ParamSet and ParamPush
void ChangeDescent()
{
    ros::NodeHandle n;
    ros::Rate r(20.0);

    ros::ServiceClient param_set_client = n.serviceClient<mavros_msgs::ParamSet>
        ("mavros/param/set");
    ros::ServiceClient param_push_client = n.serviceClient<mavros_msgs::ParamPush>
        ("mavros/param/push");
}

```

```

ValueOfParameter.real = fDescent;
ValueOfParameter.integer = 0;
SetParameter.request.value = ValueOfParameter;
SetParameter.request.param_id = "MPC_Z_VEL_MAX_DN";

    SetParameterSuccess = param_set_client.call(SetParameter);
    PushParameterSuccess = param_push_client.call(PushParameter);

    if( SetParameterSuccess && PushParameterSuccess )
    {
        printf("Velocity descent SUCCESSFULLY UPDATED with: %.1f \n", fDescent);
        printf("\n");
        StatusMsg.data = "Velocity descent SUCCESSFULLY UPDATED";
    }
    bStatus = true;
    //    bVelCallback = false;
}
else
{
    printf("Velocity descent UPDATE FAILED \n");
    printf("\n");
    StatusMsg.data = "Velocity descent UPDATED FAILED";
}
bStatus = true;
}

    ros::spinOnce();
    r.sleep();
}

//Callback for the altitude of the aircraft
void AltitudeCallback(const std_msgs::Float64::ConstPtr& msg)
{
    RelativeAltitude = msg->data;
    if ((RelativeAltitude > 1.5) && (bTakeoff != true))
    {
        bTakeoff = true;
        printf("Height reached, therefore takeoff has commenced \n");

        StatusMsg.data = "Height reached, therefore takeoff has commenced";
        bStatus = true;
    }

    if ((bOffboard == true) && (bTakeoff == true) && (RelativeAltitude < 0.2) &&
(bLandAlreadyActivated == false))
    {
        bActivateDisarm = true;
        bLandAlreadyActivated = true;
    }
}
}

```

```

//Callback for the local pose of the aircraft
void LocalposeCallback(const geometry_msgs::PoseStamped::ConstPtr &msg)
{
    local_pose=*msg;
}

//Update the aircraft's position
void MoveAircraft(double x, double y, double z)
{
    /*    final_pose.pose.position.x = x;
    final_pose.pose.position.y = y;
    final_pose.pose.position.z = z;
    */
    DeterminedAircraftPoseMsg.data[4] = x;
    DeterminedAircraftPoseMsg.data[5] = y;
    DeterminedAircraftPoseMsg.data[6] = z;
}

//Callback for the IMU of the aircraft
void imuCallback(const sensor_msgs::Imu::ConstPtr &msg)
{
    double x,y,z,w;

    x=msg->orientation.x;
    y=msg->orientation.y;
    z=msg->orientation.z;
    w=msg->orientation.w;

    IMUQuat = Eigen::Quaternionf(w,x,y,z);

    R = IMUQuat.normalized().toRotationMatrix();
}

//Callback for the flightmode received from the talker_flightmode.cpp node
void FlightmodeCallback(const std_msgs::String::ConstPtr& msg)
{
    FlightmodeMsg.data = msg->data.c_str();

    FlightmodeTemp = FlightmodeMsg.data;
    istringstream StrStreamFlightmode(FlightmodeTemp);

    StrStreamFlightmode >> sFlightmode;

    if (sFlightmode == "offboard")
    {
        bOffboard = true;
        bStabilize = false;
        printf("Flight mode received is: %s \n", sFlightmode.c_str());
    }
}

```

```

        StatusMsg.data = "The flight mode received is: offboard";
        bStatus = true;
    }
    else if (sFlightmode == "stabilized")
    {
        bOffboard = false;
        bStabilize = true;
        printf("Flight mode received is: %s \n", sFlightmode.c_str());
        StatusMsg.data = "The flight mode received is: stabilized";
        bStatus = true;
    }
}

//Callback for the coordinates received from the talker_coordinates.cpp node
void CoordinatesCallback(const std_msgs::String::ConstPtr& msg)
{
    CoordinatesMsg.data = msg->data.c_str();

    if (bManual == true)
    {
        CoordinatesTemp = CoordinatesMsg.data;
        istringstream StrStreamCoordinates(CoordinatesTemp);

        StrStreamCoordinates >> sAxis;
        StrStreamCoordinates >> sPos;

        stringstream geek(sPos);
        geek >> fPos;

        printf("The axis received: %s \n", sAxis.c_str());
        printf("The position received: %.8f \n", fPos);
        printf("\n");
        StatusMsg.data = "coordinates received";
        bStatus = true;

        if (sAxis == "x")
        {
            DeterminedAircraftPoseMsg.data[4] = fPos;
        }
        else if (sAxis == "y")
        {
            DeterminedAircraftPoseMsg.data[5] = fPos;
        }
        else if (sAxis == "z")
        {
            DeterminedAircraftPoseMsg.data[6] = fPos;
        }
    }
}

```

```

    else
    {
        printf("You entered an incorrect axis.\n");
        StatusMsg.data = "You entered an incorrect axis";
        bStatus = true;
    }
}

//Callback for the multiple coordinates received from the boundingboxmove.cpp node
void MultipleCoordinatesCallback(const std_msgs::String::ConstPtr& msg)
{
    MultipleCoordinatesMsg.data = msg->data.c_str();

    MultipleCoordinatesTemp = MultipleCoordinatesMsg.data;
    istringstream StrStreamMultipleCoordinates(MultipleCoordinatesTemp);

    StrStreamMultipleCoordinates >> sFollowY;
    StrStreamMultipleCoordinates >> sFollowX;

    stringstream geek1(sFollowY);
    geek1 >> fFollowY;

    stringstream geek2(sFollowX);
    geek2 >> fFollowX;

    bCoordinatesReceived = true;
}

//Converting the ArUco marker's quaternion to be used within the script
Eigen::Quaternionf ConvertArucoQuaternion(Eigen::Quaternionf q1)
{
    float w1 = q1.w();
    float x1 = q1.x();
    float y1 = q1.y();
    float z1 = q1.z();

    /////////// Rearranging negative signs ///////////
    if ((w1 < 0) && (y1 < 0))
    {
        w1 = -w1;
        x1 = -x1;
        y1 = -y1;
        z1 = -z1;
    }
    else if ((w1 < 0) && (z1 < 0))
    {
        w1 = -w1;
        x1 = -x1;

```

```

    y1 = -y1;
    z1 = -z1;
}

//////////subtracting euler z by 180 and invert y axis//////////

    if ((z1 < 0) && (w1 <= y1)) // clockwise (angle <= 90) in degrees
    {
        x1 = y1;
        z1 = w1;
        w1 = x1;
        y1 = z1;

        y1 = -y1;
        z1 = -z1;
    }
    else if ((z1 < 0) && (w1 > y1)) // clockwise (90 < angle < 180)
    {
        x1 = -y1;
        z1 = -w1;
        w1 = x1;
        y1 = z1;

        w1 = -w1;
        x1 = -x1;
    }
    else if ((z1 >= 0) && (w1 > z1)) // anticlockwise (90 < angle < 180)
    {
        x1 = -y1;
        z1 = -w1;
        w1 = x1;
        y1 = z1;

        w1 = -w1;
        x1 = -x1;
    }
    else if ((z1 >= 0) && (w1 <= z1)) // anticlockwise (angle <= 90)
    {
        x1 = -y1;
        z1 = -w1;
        w1 = x1;
        y1 = z1;

        y1 = -y1;
        z1 = -z1;
    }

    return Eigen::Quaternionf(w1,x1,y1,z1);
}

```



```

//Callback for the quaternion received from the QuatFromUser topic
void QuatCallback(const std_msgs::String::ConstPtr& msg)
{
    QuatMsg.data = msg->data.c_str();

    QuatTemp = QuatMsg.data;
    stringstream StrStreamQuat(QuatTemp);

    StrStreamQuat >> sQuatW;
    StrStreamQuat >> sQuatX;
    StrStreamQuat >> sQuatY;
    StrStreamQuat >> sQuatZ;

    stringstream geek1(sQuatW);
    geek1 >> fQuatW;

    stringstream geek2(sQuatX);
    geek2 >> fQuatX;

    stringstream geek3(sQuatY);
    geek3 >> fQuatY;

    stringstream geek4(sQuatZ);
    geek4 >> fQuatZ;

    //Commented out since constantly publishing in human detection system
    // printf("The w quat received: %.8f \n", fQuatW);
    // printf("The x quat received: %.8f \n", fQuatX);
    // printf("The y quat received: %.8f \n", fQuatY);
    // printf("The z quat received: %.8f \n", fQuatZ);
    // printf("\n");
    // StatusMsg.data = "Quat received";
    // bStatus = true;
    GeneralQuat = Eigen::Quaternionf(fQuatW, fQuatX, fQuatY, fQuatZ);
}

//Callback for the velocity from the VelocityFromUser topic
void VelocityCallback(const std_msgs::String::ConstPtr& msg)
{
    VelocityMsg.data = msg->data.c_str();

    VelocityTemp = VelocityMsg.data;
    stringstream StrStreamVelocity(VelocityTemp);

    StrStreamVelocity >> sVelocity;

    stringstream VelocitySS(sVelocity);
    VelocitySS >> fVelocity;
}

```

```

fVelocity = roundf(fVelocity * 10) / 10;

printf("The velocity received: %.1f \n", fVelocity);
printf("\n");
StatusMsg.data = "velocity received";
bStatus = true;

    ChangeVelocity();

}

//Callback for the descending velocity from the DescentFromUser topic
void DescentCallback(const std_msgs::String::ConstPtr& msg)
{
    DescentMsg.data = msg->data.c_str();

    DescentTemp = DescentMsg.data;
    istringstream StrStreamDescent(DescentTemp);

    StrStreamDescent >> sDescent;

    stringstream DescentSS(sDescent);
    DescentSS >> fDescent;
    fDescent = roundf(fDescent * 10) / 10;

    printf("The descent velocity received: %.1f \n", fDescent);
    printf("\n");
    StatusMsg.data = "Descent velocity received";
    bStatus = true;

    ChangeDescent();

}

//Perform Quaternion Multiplication
Eigen::Quaternionf QuaternionMultiply(Eigen::Quaternionf q1, Eigen::Quaternionf q2)
{
    float NewX1 = q1.x() * q2.w() + q1.y() * q2.z() - q1.z() * q2.y() + q1.w() * q2.x();
    float NewY1 = -q1.x() * q2.z() + q1.y() * q2.w() + q1.z() * q2.x() + q1.w() * q2.y();
    float NewZ1 = q1.x() * q2.y() - q1.y() * q2.x() + q1.z() * q2.w() + q1.w() * q2.z();
    float NewW1 = -q1.x() * q2.x() - q1.y() * q2.y() - q1.z() * q2.z() + q1.w() * q2.w();

    Quaternionf TempQuat = Eigen::Quaternionf(NewW1,NewX1,NewY1,NewZ1);

    return TempQuat;
}

//Method used to decide whether the aircraft must rotate for the landing system or for a user
input

```

```

void RotateAircraft(string sTypeOfRotation)
{
    LocalPoseQuat =
Eigen::Quaternionf(local_pose.pose.orientation.w, local_pose.pose.orientation.x, local_pose.pose.orientation.y, local_pose.pose.orientation.z);

    if (sTypeOfRotation == "marker")
    {
        ArucoQuat =
Eigen::Quaternionf(ArucoOrientation[0], ArucoOrientation[1], ArucoOrientation[2], ArucoOrientation[3]);

        NewArucoQuat = ConvertArucoQuaternion(ArucoQuat);
        FinalRotation = QuaternionMultiply(LocalPoseQuat.normalized(),
NewArucoQuat.normalized());

    }

    else if (sTypeOfRotation == "general")
    {
        FinalRotation = QuaternionMultiply(LocalPoseQuat, GeneralQuat);
        FinalRotation = FinalRotation.normalized();
    }

    DeterminedAircraftPoseMsg.data[0] = FinalRotation.w();
    DeterminedAircraftPoseMsg.data[1] = FinalRotation.x();
    DeterminedAircraftPoseMsg.data[2] = FinalRotation.y();
    DeterminedAircraftPoseMsg.data[3] = FinalRotation.z();

}

//Executes the landing system
void RepositionAircraft(bool bHoverOnly)
{
    positionaf=R*positionbe; //R is important in order to keep the drone stabilized
above the marker

//    update the position
double x=(positionaf[0]+local_pose.pose.position.x);
double y=(positionaf[1]+local_pose.pose.position.y);
double z=(positionaf[2]+local_pose.pose.position.z);

    if (bHoverOnly == false)
    {
        if (MarkerHeight > 1)
        {
            HeightWithRatio = MarkerHeight * CamRatioValue;
        }
        else
        {

```

```

        HeightWithRatio = 0.2;
    }

    if (bStartRotation == true)
    {
        if ((ros::Time::now() - rotate_time) <= ros::Duration(5.0))
        {
            if (DisplayedAlready == false)
            {
                printf("Rotation is starting soon... \n");
                StatusMsg.data = "Rotation is starting soon...";
                bStatus = true;
                DisplayedAlready = true;
            }
        }

        if (((ros::Time::now() - rotate_time) <= ros::Duration(12.0)) &&
            ((ros::Time::now() - rotate_time) > ros::Duration(5.0)))
        {
            if (DisplayedAlready == false)
            {
                printf("The aircraft has started to rotate... \n");
                StatusMsg.data = "The aircraft has started to rotate...";
                bStatus = true;
                DisplayedAlready = true;
            }

            if (bFirstRotation == true)
            {
                RotateAircraft("marker");
                bBusyRotating = true;
                bFirstRotation = false;
            }
        }
    }

    else if ((ros::Time::now() - rotate_time) > ros::Duration(12.0))
    {
        printf("12 seconds finished... \n");
        StatusMsg.data = "12 seconds finished...";
        bStatus = true;
        bBusyRotating = false;
        bStartRotation = false;
        bAllowedToLand = true;
        DisplayedAlready = false;
    }
}

```

```

    if (((HeightWithRatio > positionbe[0]) && (positionbe[0] > -HeightWithRatio)) &&
        ((HeightWithRatio > positionbe[1]) && (positionbe[1] > -HeightWithRatio)))
    {
        if (bStartTimer == false)
        {
            rotate_time = ros::Time::now();
            bStartRotation = true;
            bStartTimer = true;
            printf("Timer started...\n");
            StatusMsg.data = "Timer started...";
            bStatus = true;
        }

        if ((bBusyRotating == false) && (bAllowedToLand == true))
        {
            if(local_pose.pose.position.z >= 1)
            {
                MoveAircraft(x,y,z);

                if (DisplayedAlready == false)
                {
                    printf("Dropping altitude...\n");
                    StatusMsg.data = "Dropping altitude...";
                    bStatus = true;
                    DisplayedAlready = true;
                }
            }

            else if(local_pose.pose.position.z < 1)
            {
                MoveAircraft(x,y,-2);
            }
        }
    }

    else
    {
        if (z != 0)
        {
            MoveAircraft(x,y,local_pose.pose.position.z);
            // printf("Positioning drone above aruco marker..\n");
        }
    }

    else if (bHoverOnly == true)
    {
        MoveAircraft(x,y,MaintainHeight);
    }
}

```

```

}

//Callback for the ArUco marker's pose
void ArucoCallback(const geometry_msgs::PoseStamped::ConstPtr& msg)
{
//    converting the pose to ground frame
positionbe[1]= -(msg->pose.position.x);
positionbe[0]= -(msg->pose.position.y);
positionbe[2]= -(msg->pose.position.z);
MarkerHeight = -(msg->pose.position.z);

    ArucoOrientation[0] = msg->pose.orientation.w;
    ArucoOrientation[1] = msg->pose.orientation.x;
    ArucoOrientation[2] = msg->pose.orientation.y;
    ArucoOrientation[3] = msg->pose.orientation.z;

    if (bAutoMove == true)
    {
        if (bHover == false)
        {
            RepositionAircraft(false);
        }
        else if (bHover == true)
        {
            RepositionAircraft(true);
        }
    }
}

//Executes the human detection system
void FollowPerson()
{
    GeneralQuat = Eigen::Quaternionf(fQuatW,fQuatX,fQuatY,fQuatZ);

    bFollow = true;
    bFirstSetpoint = true;
/*
    printf("The w quat received: %.8f \n", fQuatW);
    printf("The x quat received: %.8f \n", fQuatX);
    printf("The y quat received: %.8f \n", fQuatY);
    printf("The z quat received: %.8f \n", fQuatZ);
    printf("The follow x coordinate received: %.8f \n", fFollowX);
    printf("The follow y coordinate received: %.8f \n", fFollowY);
*/
    while (bFollow == true)
    {
        if (bStartTimer == false)
        {
            follow_time = ros::Time::now();

```

```

        bStartRotation = true;
        bStartTimer = true;
        printf("Timer started...\n");
    }

    if ((ros::Time::now() - follow_time) <= ros::Duration(2.0))
    {

        if (DisplayedAlready == false)
        {
            printf("Rotation is starting soon... \n");
            StatusMsg.data = "Rotation is starting soon...";
            bStatus = true;
            DisplayedAlready = true;
        }
    }

    if (((ros::Time::now() - follow_time) <= ros::Duration(7.0)) && ((ros::Time::now() -
follow_time) > ros::Duration(2.0)))
    {
        if (DisplayedAlready == false)
        {
            printf("The aircraft has started to rotate... \n");
            StatusMsg.data = "The aircraft has started to rotate...";
            bStatus = true;
            DisplayedAlready = true;
        }

        if (bFirstRotation == true)
        {
            RotateAircraft("general");
            bBusyRotating = true;
            bFirstRotation = false;
        }
    }

    if ((ros::Time::now() - follow_time) > ros::Duration(7.0))
    {
        if ((bFirstSetpoint == true) && (bCoordinatesReceived == true))
        {
            printf("Follow coordinates uploaded... \n");
            DeterminedAircraftPoseMsg.data[4] = fFollowX;
            DeterminedAircraftPoseMsg.data[5] = fFollowY;
            DeterminedAircraftPoseMsg.data[6] = fFollowZ;

            bFollow = false;
            bStartTimer = false;
            bFirstSetpoint = false;
        }
    }

```

```

        bCoordinatesReceived = false;
    }
}

//Callback for the autoland command received from the talker_autoland.cpp node
void AutolandCallback(const std_msgs::String::ConstPtr& msg)
{
    AutolandMsg.data = msg->data.c_str();

    AutolandTemp = AutolandMsg.data;
    stringstream StrStreamAutoland(AutolandTemp);

    StrStreamAutoland >> sAutoland;

    if (sAutoland == "manual")
    {
        printf("Manual control activated \n");
        StatusMsg.data = "Manual control activated";
        bStatus = true;
        bAutoMove = false;
        bManual = true;
    }
    else if (sAutoland == "hover")
    {
        MaintainHeight = local_pose.pose.position.z;
        printf("Auto move activated \n");
        StatusMsg.data = "Auto move activated";
        bStatus = true;

        bManual = false;
        bHover = true;
        bAutoMove = true;
        bFirstRotation = true;
    }
    else if (sAutoland == "land")
    {
        MaintainHeight = local_pose.pose.position.z;

        printf("Land activated \n");
        StatusMsg.data = "Land activated";
        bStatus = true;

        bManual = false;
        bHover = false;
        bAutoMove = true;
        bFirstRotation = true;
    }
}

```



```

}
else if (sAutoland == "rotate")
{
    printf("Rotate activated \n");
    StatusMsg.data = "Rotate activated";
    bStatus = true;
    //bFirst = true;

    bManual = false;
    RotateAircraft("general");

}
else if (sAutoland == "follow")
{
    // MaintainHeight = 30;
    printf("Follow person activated \n");
    StatusMsg.data = "Follow person activated";
    bStatus = true;

    bManual = false;
    bHover = false;
    bAutoMove = false;
    bFirstRotation = true;
    FollowPerson();
}

else
{
    printf("Flight mode input incorrect. \n");
}
}

//Main method
int main(int argc, char **argv)
{
    //name assigned for the node
    ros::init(argc, argv, "determinepose");
    ros::NodeHandle nh;

    //Topics that are subscribed to
    ros::Subscriber sub_Coordinates = nh.subscribe("CoordinatesFromUser", 10,
CoordinatesCallback);
    ros::Subscriber sub_Quat = nh.subscribe("QuatFromUser", 100, QuatCallback);
    ros::Subscriber sub_Velocity = nh.subscribe("VelocityFromUser", 10, VelocityCallback);
    ros::Subscriber sub_Descent = nh.subscribe("DescentFromUser", 10, DescentCallback);
    ros::Subscriber sub_Autoland = nh.subscribe("AutolandFromUser", 10, AutolandCallback);
    ros::Subscriber sub_Flightmode = nh.subscribe("FlightmodeFromUser", 10,
FlightmodeCallback);

```

```

ros::Subscriber sub_MultipleCoordinates = nh.subscribe("/MultipleCoordinates", 10,
MultipleCoordinatesCallback);
ros::Subscriber state_sub = nh.subscribe<mavros_msgs::State>
    ("mavros/state", 10, state_cb);
ros::Subscriber imu_sub = nh.subscribe<sensor_msgs::Imu>
    ("/mavros/imu/data",10,imuCallback);
ros::Subscriber localpose_sub = nh.subscribe<geometry_msgs::PoseStamped>
    ("/mavros/local_position/pose",100,LocalposeCallback);
ros::Subscriber aruco_sub = nh.subscribe<geometry_msgs::PoseStamped>
    ("aruco_single/pose", 1000, ArucoCallback);
ros::Subscriber altitude_sub =
nh.subscribe<std_msgs::Float64>("mavros/global_position/rel_alt", 10, AltitudeCallback);

//Topics that are published to
ros::Publisher local_pos_pub = nh.advertise<geometry_msgs::PoseStamped>
    ("mavros/setpoint_position/local", 10);
ros::Publisher aircraftpose_pub = nh.advertise<std_msgs::Float64MultiArray>
    ("/determinedaircraftpose", 100);
ros::Publisher status_pub = nh.advertise<std_msgs::String>("StatusFromAircraft", 10);

//Service clients that are communicated
ros::ServiceClient arming_client = nh.serviceClient<mavros_msgs::CommandBool>
    ("mavros/cmd/arming");
ros::ServiceClient set_mode_client = nh.serviceClient<mavros_msgs::SetMode>
    ("mavros/set_mode");

ros::Rate rate(20.0);

DeterminedAircraftPoseMsg.data.clear();

    for (int i = 0; i < 7; i++)
    {
        DeterminedAircraftPoseMsg.data.push_back(0);
    }

//Wait till communication with the flight controller
while(ros::ok() && !current_state.connected)
{
    ros::spinOnce();
    rate.sleep();
}

DeterminedAircraftPoseMsg.data[0] = local_pose.pose.orientation.w;
DeterminedAircraftPoseMsg.data[1] = local_pose.pose.orientation.x;
DeterminedAircraftPoseMsg.data[2] = local_pose.pose.orientation.y;
DeterminedAircraftPoseMsg.data[3] = local_pose.pose.orientation.z;
DeterminedAircraftPoseMsg.data[4] = local_pose.pose.position.x;
DeterminedAircraftPoseMsg.data[5] = local_pose.pose.position.y;
DeterminedAircraftPoseMsg.data[6] = 2;

```

```

ChangeVelocity();
ChangeDescent();

printf("The pose may now be adjusted \n");
StatusMsg.data = "Determinepose is ready.";

while(ros::ok())
{
    //the position and orientation is published to the topic determinedaircraftpose
    aircraftpose_pub.publish(DeterminedAircraftPoseMsg);

    //variables are reset back to default when the aircraft has disarmed
    if ((!current_state.armed) && bActivatedDisarm == true)
    {
        sAxis = "z";
        sPos = "2";
        sVelocity = "1";
        sAutoland = "false";
        sDescent = "0.2";
        sFlightmode = "";
        sYaw = "0";
        sQuatX = "0";
        sQuatY = "0";
        sQuatZ = "0";
        sQuatW = "0";
        sFollowX = "0";
        sFollowY = "0";

        fVelocity = 1.5;
        fDescent = 0.2;
        fPos = 2;
        AverageHomeZ = 0;
        CamRatioValue = 0.2;
        HeightWithRatio = 0;
        MarkerHeight = 0;
        fYawDegrees = 0;
        fYawRadians = 0;
        fQuatX = 0;
        fQuatY = 0;
        fQuatZ = 0;
        fQuatW = 0;
        ArucoOrientation[0] = 0;
        ArucoOrientation[1] = 0;
        ArucoOrientation[2] = 0;
        ArucoOrientation[3] = 0;

        MaintainHeight = 0;
        MaintainX = 0;
        MaintainY = 0;
        YawEuler = 0;
    }
}

```

```

fFollowX = 0;
fFollowY = 0;
fFollowZ = 30;
RelativeAltitude = 0;

bVelCallback = false;
bAutoMove = false;
SetParameterSuccess;
PushParameterSuccess;
bTakeoff = false;
bArmed = false;
bOffboard = false;
bStabilize = false;
bStatus = false;
bRotate = false;
bFirstRotation = false;
bFirstSetpoint = false;
bManual = true;
bHover = false;
bStartTimer = false;
bStartRotation = false;
bBusyRotating = false;
bAllowedToLand = false;
bFollow = false;
bCoordinatesReceived = false;
DisplayedAlready = false;
bLandAlreadyActivated = false;

bActivateDisarm = false;

DeterminedAircraftPoseMsg.data[0] = local_pose.pose.orientation.w;
DeterminedAircraftPoseMsg.data[1] = local_pose.pose.orientation.x;
DeterminedAircraftPoseMsg.data[2] = local_pose.pose.orientation.y;
DeterminedAircraftPoseMsg.data[3] = local_pose.pose.orientation.z;

DeterminedAircraftPoseMsg.data[4] = local_pose.pose.position.x;
DeterminedAircraftPoseMsg.data[5] = local_pose.pose.position.y;
DeterminedAircraftPoseMsg.data[6] = 2;

printf("The aircraft has been disarmed and the variables have been reset.\n");
StatusMsg.data = "The aircraft has been disarmed and the variables have been reset.";

status_pub.publish(StatusMsg);
}

if (bStatus == true)
{
status_pub.publish(StatusMsg);
bStatus = false;
}

```

```
    ros::spinOnce();  
    rate.sleep();  
  
}  
  
return 0;  
}
```

Appendix 5.2: flyaircraft.cpp node

```
//relevant libraries used in this node
#include <ros/ros.h>
#include <geometry_msgs/PoseStamped.h>
#include <mavros_msgs/CommandBool.h>
#include <mavros_msgs/SetMode.h>
#include <mavros_msgs/State.h>
#include <std_msgs/Float64MultiArray.h>
#include "std_msgs/MultiArrayLayout.h"
#include "std_msgs/MultiArrayDimension.h"
#include <std_msgs/String.h>
#include <std_msgs/Float64.h>

// this allows identifiers to be used
using namespace std;

//declaring variables
mavros_msgs::State current_state;
mavros_msgs::SetMode offb_set_mode;
mavros_msgs::CommandBool arm_cmd;

std_msgs::String FlightmodeMsg;

string FlightmodeTemp;

string sFlightmode = "";

bool bOffboard = false;
bool bOffboardSet = false;
bool bStabilizeSet = false;
bool bLandSet = false;
bool bArmAircraft = false;
bool bPublish = false;
bool bTakeoff = false;
bool bAirborne = false;
bool bLandAlreadyActivated = false;
bool bActivateDisarm = false;

std_msgs::Float64MultiArray TempPose;
float ArrPose[7];

float RelativeAltitude = 0;

//Callback to receive the state of the aircraft
void state_cb(const mavros_msgs::State::ConstPtr& msg)
{
    current_state = *msg;
}
```

```

//Callback to receive the pose for the aircraft from the determinedaircraftpose topic
void AircraftPoseCallback(const std_msgs::Float64MultiArray::ConstPtr& msg)
{
    for (int i = 0; i < 7; i++)
    {
        ArrPose[i] = msg->data[i];
    }
}

//Callback to receive the altitude of the aircraft
void AltitudeCallback(const std_msgs::Float64::ConstPtr& msg)
{
    RelativeAltitude = msg->data;

    if ((RelativeAltitude > 1.5) && (bTakeoff != true))
    {
        bTakeoff = true;
    }

    if ((bOffboard == true) && (bTakeoff == true) && (RelativeAltitude < 0.2) &&
(bLandAlreadyActivated == false))
    {
        bLandSet = true;
        bOffboard = false;
        bActivateDisarm = true;
    }
}

//Callback to receive the flight mode from the talker_flightmode.cpp node
void FlightmodeCallback(const std_msgs::String::ConstPtr& msg)
{
    FlightmodeMsg.data = msg->data.c_str();

    FlightmodeTemp = FlightmodeMsg.data;
    istringstream StrStreamFlightmode(FlightmodeTemp);

    StrStreamFlightmode >> sFlightmode;

    if (sFlightmode == "offboard")
    {
        bOffboard = true;
        bOffboardSet = true;
        bStabilizeSet = false;
        bLandSet = false;
        bArmAircraft = true;
        bPublish = true;
        bLandAlreadyActivated = false;
        printf("Flight mode received is: %s \n", sFlightmode.c_str());
    }
}

```

```

}
else if (sFlightmode == "stabilized")
{
    bOffboard = false;
    bOffboardSet = false;
    bStabilizeSet = true;
    bLandSet = false;
    bArmAircraft = true;
    bPublish = false;
    printf("Flight mode received is: %s \n", sFlightmode.c_str());
}
else if (sFlightmode == "land")
{
    bOffboard = false;
    bOffboardSet = false;
    bStabilizeSet = false;
    bLandSet = true;
    bPublish = false;
    printf("Flight mode received is: %s \n", sFlightmode.c_str());
}
}

//Main Method
int main(int argc, char **argv)
{
//name assigned for the node
    ros::init(argc, argv, "flyaircraft");
    ros::NodeHandle nh;

//Topics that are subscribed to
    ros::Subscriber state_sub = nh.subscribe<mavros_msgs::State>
        ("mavros/state", 10, state_cb);
    ros::Subscriber sub_aircraftpose =
nh.subscribe<std_msgs::Float64MultiArray>("/determinedaircraftpose", 100,
AircraftPoseCallback);
    ros::Subscriber sub_Flightmode = nh.subscribe("FlightmodeFromUser", 10,
FlightmodeCallback);
    ros::Subscriber altitude_sub =
nh.subscribe<std_msgs::Float64>("mavros/global_position/rel_alt", 10, AltitudeCallback);

//Topics that are published to
    ros::Publisher local_pos_pub = nh.advertise<geometry_msgs::PoseStamped>
        ("mavros/setpoint_position/local", 10);

//Service clients that are communicated
    ros::ServiceClient arming_client = nh.serviceClient<mavros_msgs::CommandBool>
        ("mavros/cmd/arming");
    ros::ServiceClient set_mode_client = nh.serviceClient<mavros_msgs::SetMode>
        ("mavros/set_mode");

```



```

//the setpoint publishing rate MUST be faster than 2Hz
ros::Rate rate(20.0);

printf("flyaircraft has started...\n");

// wait till communication with the flight controller
while(ros::ok() && !current_state.connected){
    ros::spinOnce();
    rate.sleep();
}

geometry_msgs::PoseStamped pose;
pose.pose.position.x = 0;
pose.pose.position.y = 0;
pose.pose.position.z = 2;

//send a few setpoints before the aircraft may be used in offboard mode
for(int i = 100; ros::ok() && i > 0; --i){
    local_pos_pub.publish(pose);
    ros::spinOnce();
    rate.sleep();
}

printf("100 setpoints published\n");

offb_set_mode.request.custom_mode = "OFFBOARD";

arm_cmd.request.value = true;

ros::Time last_request = ros::Time::now();

    printf("starting offboard and arming sequence...\n");

while(ros::ok())
{

    if (bOffboardSet == true)
    {
        pose.pose.position.x = 0;
        pose.pose.position.y = 0;
        pose.pose.position.z = 2;

//send a few setpoints before starting
        for(int i = 100; ros::ok() && i > 0; --i)
        {

            local_pos_pub.publish(pose);
            ros::spinOnce();
            rate.sleep();

```

```

    }
    //Switches to offboard mode
    printf("Inside OffboardSet \n");
    offb_set_mode.request.custom_mode = "OFFBOARD";
    set_mode_client.call(offb_set_mode);
    offb_set_mode.response.mode_sent;
    arm_cmd.request.value = true;
    bOffboardSet = false;
    ros::Time last_request = ros::Time::now();
}

//Switches to Land flight mode when altitude point has been reached
else if (bLandSet == true)
{
    offb_set_mode.request.custom_mode = "AUTO.LAND";
    set_mode_client.call(offb_set_mode);
    offb_set_mode.response.mode_sent;
    arm_cmd.request.value = true;
    bLandSet = false;
    bLandAlreadyActivated = true;
    printf("AUTO.LAND has been requested... \n");
}

//Code used specifically to use this node on Gazebo simulation
//Start of simulation code

if( current_state.mode != "OFFBOARD" &&
    (ros::Time::now() - last_request > ros::Duration(5.0)) && bOffboard == true)
{
    if( set_mode_client.call(offb_set_mode) &&
        offb_set_mode.response.mode_sent)
    {
        ROS_INFO("Offboard enabled");
    }
    last_request = ros::Time::now();
}

else
{
    if( !current_state.armed &&
        (ros::Time::now() - last_request > ros::Duration(5.0)) && bArmAircraft == true)
    {
        printf("Inside Attempting to arm... \n");
        if( arming_client.call(arm_cmd) &&
            arm_cmd.response.success)
        {
            ROS_INFO("Vehicle armed");
            bArmAircraft = false;
            //      bPublish = true;

```

```

        }
        last_request = ros::Time::now();
    }
}

//End of simulation code

/*
//Code used specifically to use this node on the actual aircraft
//Start of actual aircraft code
    if( current_state.mode != "OFFBOARD" &&
        (ros::Time::now() - last_request > ros::Duration(5.0)))
    {
        if( current_state.mode == "OFFBOARD")
        {
            ROS_INFO("Offboard enabled");
        }
        last_request = ros::Time::now();
    }

    else
    {
        //    if( !current_state.armed && (ros::Time::now() - last_request > ros::Duration(5.0))
&& bArmAircraft == true)
            if( !current_state.armed &&
                (ros::Time::now() - last_request > ros::Duration(5.0)) && (bOffboard == true)
&& current_state.mode == "OFFBOARD")
            {
                printf("Inside Arming sequence \n");
                if( arming_client.call(arm_cmd) &&
                    arm_cmd.response.success)
                {
                    ROS_INFO("Vehicle armed");
                    //    bArmAircraft = false;
                    bPublish = true;
                }
                last_request = ros::Time::now();
            }
    }
}
*/
//End of actual aircraft code

    if ((bPublish == true) && (current_state.mode != "AUTO.LAND"))
    {
        pose.pose.orientation.w = ArrPose[0];
        pose.pose.orientation.x = ArrPose[1];
        pose.pose.orientation.y = ArrPose[2];
        pose.pose.orientation.z = ArrPose[3];
    }
}

```

```

        pose.pose.position.x = ArrPose[4];
        pose.pose.position.y = ArrPose[5];
        pose.pose.position.z = ArrPose[6];

        //publish the pose to the aircraft via MAVROS
        local_pos_pub.publish(pose);
    }

    if (!(current_state.armed) && bActivatedDisarm == true)
    {
        //resets all the variables
        bOffboard = false;
        bOffboardSet = false;
        bStabilizeSet = false;
        bLandSet = false;
        bArmAircraft = false;
        bPublish = false;
        bTakeoff = false;
        bAirborne = false;
        bLandAlreadyActivated = false;
        bActivatedDisarm = false;

        printf("Aircraft has been disarmed \n");
    }

    ros::spinOnce();
    rate.sleep();
}

return 0;
}

```

Appendix 5.3: talker_autoland.cpp node

```
//relevant libraries used in this node
#include "ros/ros.h"
#include "std_msgs/String.h"
#include <iostream>
#include <string>
#include <sstream>

// this allows identifiers to be used
using namespace std;

//Declared variables
string sAutoland = "false"; //DEFAULT VELOCITY

//Main Method
int main(int argc, char **argv)
{
//name assigned for the node
ros::init(argc, argv, "talker_autoland");

ros::NodeHandle nh;

//Topics that are published
ros::Publisher autoland_pub = nh.advertise<std_msgs::String>("AutolandFromUser", 10);

ros::Rate loop_rate(20);

int count = 0;
while (ros::ok())
{

std_msgs::String msg;

std::stringstream ss;

//Receives the user's input for the type of autonomous flight
cout << "Please either enter 'manual' , 'hover' , 'land' , 'follow' or 'rotate':" << endl;

cin >> sAutoland;

ss << sAutoland;

printf("\n");
printf("The autoland you have chosen is: %s \n", sAutoland.c_str());
msg.data = ss.str();
printf("\n");

//Publishes the user's input
autoland_pub.publish(msg);
```

```
    ros::spinOnce();  
  
    loop_rate.sleep();  
  
}  
  
return 0;  
}
```

Appendix 5.4: talker_coordinates.cpp node

```
//relevant libraries used in this node
#include "ros/ros.h"
#include "std_msgs/String.h"
#include <iostream>
#include <string>
#include <sstream>

// this allows identifiers to be used
using namespace std;

//Variables declared
string sAxis = "z";
string sPos = "2";
float fPos = 2;

//Main method
int main(int argc, char **argv)
{
    //name assigned for the node
    ros::init(argc, argv, "talker_coordinates");

    ros::NodeHandle nh;

    //Topics that are published to
    ros::Publisher coordinates_pub = nh.advertise<std_msgs::String>("CoordinatesFromUser", 10);

    ros::Rate loop_rate(20);

    int count = 0;
    while (ros::ok())
    {

        std_msgs::String msg;

        std::stringstream ss;

        //Receives the coordinate from the user
        cout << "Please enter axis and position value (e.g. x 2):" << endl;

        cin >> sAxis >> sPos;

        ss << sAxis + " " + sPos;

        stringstream geek(sPos);

        geek >> fPos;

        printf("\n");
```

```
    printf("The axis you have chosen: %s \n", sAxis.c_str());
    printf("The value you have chosen: %.8f \n", fPos);
    printf("\n");

    msg.data = ss.str();

    //Publishes the user's input
    coordinates_pub.publish(msg);

    ros::spinOnce();

    loop_rate.sleep();

}

return 0;
}
```


Appendix 5.5: talker_descent.cpp node

```
//relevant libraries used in this node
#include "ros/ros.h"
#include "std_msgs/String.h"
#include <iostream>
#include <string>
#include <sstream>

// this allows identifiers to be used
using namespace std;

//Variables declared
string sVelocity = "1"; //DEFAULT VELOCITY

//Main Method
int main(int argc, char **argv)
{

    //name assigned for the node
    ros::init(argc, argv, "talker_descent");

    ros::NodeHandle nh;

    //Topics that are published
    ros::Publisher descent_pub = nh.advertise<std_msgs::String>("DescentFromUser", 10);

    ros::Rate loop_rate(20);

    int count = 0;
    while (ros::ok())
    {

        std_msgs::String msg;

        std::stringstream ss;

        //Receives the descending velocity from the user
        cout << "Please enter a velocity in m/s with decimal (e.g. 1.0):" << endl;

        cin >> sVelocity;

        ss << sVelocity;

        printf("\n");
        printf("The velocity you have chosen: %s m/s\n", sVelocity.c_str());
        msg.data = ss.str();
        printf("\n");

        //Publishes the descending velocity from the user
```

```
descent_pub.publish(msg);  
  
ros::spinOnce();  
  
loop_rate.sleep();  
  
}  
  
return 0;  
}
```

Appendix 5.6: talker_velocity.cpp node

```
//relevant libraries used in this node
#include "ros/ros.h"
#include "std_msgs/String.h"
#include <iostream>
#include <string>
#include <sstream>

// this allows identifiers to be used
using namespace std;

//Variables declared
string sVelocity = "12"; //DEFAULT VELOCITY

int main(int argc, char **argv)
{
//name assigned for the node
ros::init(argc, argv, "talker_velocity");

ros::NodeHandle nh;

//Topics that are published to
ros::Publisher velocity_pub = nh.advertise<std_msgs::String>("VelocityFromUser", 10);

ros::Rate loop_rate(20);

int count = 0;
while (ros::ok())
{

std_msgs::String msg;

std::stringstream ss;

//Receives the horizontal velocity from the user
cout << "Please enter a velocity in m/s with decimal (e.g. 12.0):" << endl;

cin >> sVelocity;

ss << sVelocity;

printf("\n");
printf("The velocity you have chosen: %s m/s\n", sVelocity.c_str());
msg.data = ss.str();
printf("\n");

//Publishes the velocity from the user
velocity_pub.publish(msg);
```

```
    ros::spinOnce();

    loop_rate.sleep();
}

return 0;
}
```

Appendix 5.7: talker_flightmode.cpp node

```
//relevant libraries used in this node
#include "ros/ros.h"
#include "std_msgs/String.h"
#include <iostream>
#include <string>
#include <sstream>

// this allows identifiers to be used
using namespace std;

//Variables declared
string sFlightmode = "false";

//Main Method
int main(int argc, char **argv)
{
//name assigned for the node
ros::init(argc, argv, "talker_flightmode");

ros::NodeHandle nh;

//Topics that are published
ros::Publisher flightmode_pub = nh.advertise<std_msgs::String>("FlightmodeFromUser", 10);

ros::Rate loop_rate(20);

int count = 0;
while (ros::ok())
{

std_msgs::String msg;

std::stringstream ss;

//Receives the flight mode from the user
cout << "Please enter flightmode (e.g. offboard or stabilized):" << endl;

cin >> sFlightmode;

ss << sFlightmode;

printf("\n");
printf("The flightmode you have chosen is: %s \n", sFlightmode.c_str());
msg.data = ss.str();
printf("\n");

//Publishes the flight mode from the user
flightmode_pub.publish(msg);
```

```
    ros::spinOnce();  
  
    loop_rate.sleep();  
  
}  
  
return 0;  
}
```

Appendix 5.8: talker_quat.cpp node

```
//relevant libraries used in this node
#include "ros/ros.h"
#include "std_msgs/String.h"
#include <iostream>
#include <string>
#include <sstream>

// this allows identifiers to be used
using namespace std;

//Variables declared
string sX = "0";
string sY = "0";
string sZ = "0";
string sW = "0";

//Main Method
int main(int argc, char **argv)
{
    //name assigned for the node
    ros::init(argc, argv, "talker_quat");

    ros::NodeHandle nh;
    //Topics that are published
    ros::Publisher quat_pub = nh.advertise<std_msgs::String>("QuatFromUser", 100);

    ros::Rate loop_rate(20);

    int count = 0;
    while (ros::ok())
    {

        std_msgs::String msg;

        std::stringstream ss;

//Receives a quaternion from the user
        cout << "Please a quaternion (format is: w x y z):" << endl;

        cin >> sW >> sX >> sY >> sZ;

        ss << sW + " " + sX + " " + sY + " " + sZ;

        //        stringstream geek(sPos);

        //        geek >> fPos;

        printf("\n");
    }
}
```

```
    printf("The w you have chosen: %s \n", sW.c_str());
    printf("The x you have chosen: %s \n", sX.c_str());
    printf("The y you have chosen: %s \n", sY.c_str());
    printf("The z you have chosen: %s \n", sZ.c_str());

//    printf("The value you have chosen: %.8f \n", fPos);
    printf("\n");

    msg.data = ss.str();

//Publishes the quaternion from the user
    quat_pub.publish(msg);

    ros::spinOnce();

    loop_rate.sleep();

}

return 0;
}
```


Appendix 5.9: listener_status.cpp node

```
//relevant libraries used in this node
#include "ros/ros.h"
#include "std_msgs/String.h"

// this allows identifiers to be used
using namespace std;

//Variables declared
std_msgs::String StatusMsg;

string sStatus = "null";

string StatusTemp;

//Callback to receive messages from the StatusFromAircraft topic
void StatusCallback(const std_msgs::String::ConstPtr& msg)
{
    StatusMsg.data = msg->data.c_str();

    StatusTemp = StatusMsg.data;

    printf("%s \n", StatusTemp.c_str());
    printf("\n");
}

//Main Method
int main(int argc, char **argv)
{
    //name assigned for the node
    ros::init(argc, argv, "listener_status");

    ros::NodeHandle nh;

    //Topics that are subscribed to
    ros::Subscriber sub_Coordinates = nh.subscribe("StatusFromAircraft", 10, StatusCallback);

    ros::Rate r(20);

    while (ros::ok())
    {

        ros::spinOnce();
        r.sleep();
    }
    return 0;
}
```

Appendix 5.10: boxinfo.cpp node

```
//relevant libraries used in this node
#include "ros/ros.h"
#include "std_msgs/String.h"
#include "darknet_ros_msgs/BoundingBoxes.h"
#include <math.h>
#include <sstream>
#include <iostream>
#include <iterator>
#include <string>
#include <boost/range.hpp>

// this allows identifiers to be used
using namespace std;

//Variables declared
darknet_ros_msgs::BoundingBoxes BoxMsg;

int xmax;
int xmin;
int ymax = 380;
int ymin;
int xavg;
int yavg;
int xdistancetomid;
int ydistancetomid;
int yDifference;
int boundingboxsize;

//webcam resolution
//int camerawidth = 640;
//int cameraheight = 480;

//DJI video Footage resolution
int camerawidth = 1920;
int cameraheight = 1080;

int camerawidthhalved = camerawidth / 2;
int cameraheighthalved = cameraheight / 2;

//double hfov = 70.42; //logitech c920 webcam
//double vfov = 43.3; //logitech c920 webcam

double hfov = 76.25; //DJI Phantom 4 Pro (calculated)
double vfov = 47.64; //DJI Phantom 4 Pro (calculated)

double AngleToRotate;

float AltitudeSet = 10;
```

```

float VFOV;
float AngleToFirstPoint;
float AngleToFinalPoint;
float GroundDistanceToFirstPoint;
float GroundDistanceToLastPoint;
float MaxDistance;
float OppAngle;
float AngleOnStraightLine;
float AngleForMainTriangle;
float Hypotenuse;
float ymaxDifference;
float HypotenuseForLinearization;
float DistanceBetweenAircraftAndPerson;
float DistanceToMove;
float yextra = 0;
float ywiththequation;
float xavgpastmidpoint;
float abovexavg

bool PubOpen = false;
//bool PubOpen = true;

std_msgs::String msgAngle;
std_msgs::String msgDistance;

//Callback to receive the bounding box information obtained by using the darknet_ros package
void BoxesCallback(const darknet_ros_msgs::BoundingBoxes::ConstPtr& msg)
{
    boundingboxsize = boost::size(msg->bounding_boxes);
    printf("Size of the array: %i \n", boundingboxsize);

    for (int i = 0; i < boundingboxsize;i++)
    {
        if (msg->bounding_boxes[i].Class == "person")
        {
            xmax = msg->bounding_boxes[0].xmax;
            xmin = msg->bounding_boxes[0].xmin;
            ymax = msg->bounding_boxes[0].ymax;
            ymin = msg->bounding_boxes[0].ymin;

            printf("xmax: %i \n", xmax);
            printf("xmin: %i \n", xmin);
            printf("ymax: %i \n", ymax);
            printf("ymin: %i \n", ymin);

            xavg = (xmax + xmin) / 2;
            yavg = (ymax + ymin) / 2;

            printf("xavg: %i \n", xavg);
            printf("yavg: %i \n", yavg);
        }
    }
}

```

```

xdistancetomid = (camerawidth/2) - xavg;
ydistancetomid = (camerawidth/2) - yavg;

//determining which direction and angle to rotate the aircraft based on the the position
of the bounding box on the video feed
    if (xdistancetomid < 0)
    {
        printf("rotate right by %i pixels\n", -xdistancetomid);
        AngleToRotate = ((hfov / 2) * (xdistancetomid) /
camerawidthhalved);

        printf("rotate right by %.2f degrees \n", -AngleToRotate);

    }
    else
    {
        printf("rotate left by %i pixels\n", xdistancetomid);
        AngleToRotate = ((hfov / 2) * (xdistancetomid) /
camerawidthhalved);

        printf("rotate left by %.2f degrees \n", AngleToRotate);

    }

/*    if (ydistancetomid < 0)
    {
        printf("move camera up by %i pixels\n", -ydistancetomid);
    }
    else
    {
        printf("move camera down by %i pixels\n", ydistancetomid);
    }
*/

//fixes the ground distance, as mentioned in section

    if ((xavg - camerawidthhalved) < 0)
    {
        xavgpastmidpoint = camerawidthhalved + (camerawidthhalved - xavg);
    }
    else
    {
        xavgpastmidpoint = xavg;
    }

    if (ymax < 500)
    {
        yextra = 0.1793*xavgpastmidpoint - 187.81;
    }
    else if ((ymax >= 500) && (ymax < 750))
    {
        yextra = 0.0936*xavgpastmidpoint - 104.75;
    }

```

```

    }
    else if (ymax >= 750)
    {
        yextra = 0.3233*xavgpastmidpoint - 322.05;
    }

    //Calculates the ground distance
    ymaxDifference = cameraheight - (ymax + yextra);
    HypotenuseForLinearization = ymaxDifference / (sin(AngleForMainTriangle * M_PI
/ 180));
    DistanceToMove = (HypotenuseForLinearization / Hypotenuse) * MaxDistance;
    DistanceBetweenAircraftAndPerson = GroundDistanceToFirstPoint + DistanceToMove;

    printf("Distance between the aircraft and the person: %.2f metres\n",
DistanceBetweenAircraftAndPerson);
    printf("Distance aircraft needs to move: %.2f \n", DistanceToMove);
    printf("\n");

    PubOpen = true;
}
else
{
    printf("An object was detected, but not a human. \n");
    printf("\n");
}
}
}

//Main method
int main(int argc, char **argv)
{
    //name assigned for the node
    ros::init(argc, argv, "boxinfo");

    ros::NodeHandle nh;

    //Topics that are subscribed to
    ros::Subscriber sub_Box = nh.subscribe("/darknet_ros/bounding_boxes", 10, BoxesCallback);
    //Topics that are published
    ros::Publisher angle_pub = nh.advertise<std_msgs::String>("/AngleFromBoundingBox", 10);
    ros::Publisher distance_pub = nh.advertise<std_msgs::String>("/DistanceFromBoundingBox",
10);
    ros::Rate r(10);

    //Calculations part of determining the ground distance
    AngleToFirstPoint = 45 - (vfov/2);
    AngleToFinalPoint = 45 + (vfov/2);
    GroundDistanceToFirstPoint = AltitudeSet * tan(AngleToFirstPoint * M_PI / 180);
    GroundDistanceToLastPoint = AltitudeSet * tan(AngleToFinalPoint * M_PI / 180);
    MaxDistance = GroundDistanceToLastPoint - GroundDistanceToFirstPoint;

```

```

OppAngle = 180 - 90 - AngleToFirstPoint;
AngleOnStraightLine = 180 - OppAngle;
AngleForMainTriangle = 180 - AngleOnStraightLine - vfov;
Hypotenuse = cameraheight / (sin(AngleForMainTriangle * M_PI / 180));

while (ros::ok())
{
    if (PubOpen == true)
    {
        std::stringstream ss1;
        ss1 << AngleToRotate;
        msgAngle.data = ss1.str();
        //calculated angle to rotate the aircraft is published
        angle_pub.publish(msgAngle);

        std::stringstream ss2;
        ss2 << DistanceToMove;
        msgDistance.data = ss2.str();
        //calculated ground distance to move the aircraft is published
        distance_pub.publish(msgDistance);

        printf("Angle and distance determined. \n");
        printf("\n");

        PubOpen = false;
    }
    ros::spinOnce();
    r.sleep();
}

return 0;
}

```

Appendix 5.11: boundingboxmove.cpp node

```
//relevant libraries used in this node
#include "ros/ros.h"
#include <string>
#include <math.h>
#include <std_msgs/String.h>
#include <iostream>
#include <sstream>
#include <boost/lexical_cast.hpp>
#include <geometry_msgs/PoseStamped.h>

// this allows identifiers to be used
using namespace std;

//Declared variables
float fAngleAircraft; //angle that the drone is facing after it has rotated to make the person
in the centre of the screen (angle must be obtained from the aircraft)

//negative angle means rotation is clockwise

float fAngle0;
float fDistance;
float fCurrentX = 5;
float fCurrentY = 2;
float fAdjacent;
float fOpposite;
float fNewX;
float fNewY;
float fAngleReceived;
float fDistanceReceived;

bool bAngle = false;
bool bDistance = false;

string sAngle;
string sDistance;
string sTemp;
string DistanceTemp;
string AngleTemp;

std_msgs::String DistanceMsg;
std_msgs::String AngleMsg;
std_msgs::String msgCoordinates;

std::string sCoordinates;

geometry_msgs::PoseStamped local_pose;

//Callback to receive the ground distance from the boxinfo.cpp node
```

```

void DistanceCallback(const std_msgs::String::ConstPtr& msg)
{
    DistanceMsg.data = msg->data.c_str();

    DistanceTemp = DistanceMsg.data;
    istringstream StrStreamDistance(DistanceTemp);

    StrStreamDistance >> sDistance;

    stringstream DistanceSS(sDistance);
    DistanceSS >> fDistanceReceived;

// printf("Distance received: %.2f \n",fDistanceReceived);

    bDistance = true;
}

//Callback to receive the converted aircraft's angle from the QuatToEuler.py node
void AngleCallback(const std_msgs::String::ConstPtr& msg)
{
    AngleMsg.data = msg->data.c_str();

    AngleTemp = AngleMsg.data;
    istringstream StrStreamAngle(AngleTemp);

    StrStreamAngle >> sAngle;

    stringstream AngleSS(sAngle);
    AngleSS >> fAngleReceived;

// printf("Angle received: %.2f \n",fAngleReceived);

    bAngle = true;
}

//Callback to receive the local pose of the aircraft
void LocalposeCallback(const geometry_msgs::PoseStamped::ConstPtr &msg)
{
    local_pose=*msg;
}

//Method to calculate the coordinates the aircraft needs to fly towards
void CoordinatesForAircraft()
{
    fAngleAircraft = fAngleReceived; //negative means aircraft rotates clockwise
    fDistance = fDistanceReceived;
    fCurrentY = local_pose.pose.position.y;
    fCurrentX = local_pose.pose.position.x;

    printf("Angle received: %.2f \n",fAngleAircraft);
}

```



```

printf("Distance received: %.2f \n", fDistance);
printf("Aircraft's current y co-ordinate: %.2f \n", fCurrentY);
printf("Aircraft's current x co-ordinate: %.2f \n", fCurrentX);
printf("\n");

if (fDistance >= 0)
{
    if ((fAngleAircraft >= -90) && (fAngleAircraft < 0)) //Example 1
    {
        fAngle0 = 90 -(-fAngleAircraft);
        printf("fAngle0: %.2f \n", fAngle0);
        fAdjacent = cos(fAngle0 * M_PI / 180) * fDistance;
        fOpposite = sin(fAngle0 * M_PI / 180) * fDistance;

        printf("fAdjacent: %.2f \n", fAdjacent);
        printf("fOpposite: %.2f \n", fOpposite);

        fNewY = fCurrentY - fAdjacent;
        fNewX = fCurrentX + fOpposite;
    }

    else if ((fAngleAircraft >= -180) && (fAngleAircraft < -90)) //Example 2
    {
        fAngle0 = (-fAngleAircraft) - 90;
        printf("fAngle0: %.2f \n", fAngle0);
        fAdjacent = cos(fAngle0 * M_PI / 180) * fDistance;
        fOpposite = sin(fAngle0 * M_PI / 180) * fDistance;

        printf("fAdjacent: %.2f \n", fAdjacent);
        printf("fOpposite: %.2f \n", fOpposite);

        fNewY = fCurrentY - fAdjacent;
        fNewX = fCurrentX - fOpposite;
    }

    else if ((fAngleAircraft <= 180) && (fAngleAircraft > 90)) //Example 3
    {
        fAngle0 = fAngleAircraft - 90;
        printf("fAngle0: %.2f \n", fAngle0);
        fAdjacent = cos(fAngle0 * M_PI / 180) * fDistance;
        fOpposite = sin(fAngle0 * M_PI / 180) * fDistance;

        printf("fAdjacent: %.2f \n", fAdjacent);
        printf("fOpposite: %.2f \n", fOpposite);

        fNewY = fCurrentY + fAdjacent;
        fNewX = fCurrentX - fOpposite;
    }

    else if ((fAngleAircraft <= 90) && (fAngleAircraft >= 0)) //Example 4

```

```

{
    fAngle0 = 90 - fAngleAircraft;
    printf("fAngle0: %.2f \n", fAngle0);
    fAdjacent = cos(fAngle0 * M_PI / 180) * fDistance;
    fOpposite = sin(fAngle0 * M_PI / 180) * fDistance;

    printf("fAdjacent: %.2f \n", fAdjacent);
    printf("fOpposite: %.2f \n", fOpposite);

    fNewY = fCurrentY + fAdjacent;
    fNewX = fCurrentX + fOpposite;
}

printf("New Y co-ordinate: %.2f \n", fNewY);
printf("New X co-ordinate: %.2f \n", fNewX);
printf("Therefore, (y;x) = ( %.2f ; %.2f ) \n", fNewY, fNewX);
printf("\n");
}

else if (fDistance < 0)
{
// printf ("fDistance is negative \n");
if ((fAngleAircraft >= -90) && (fAngleAircraft < 0)) //Example 5
{
    fAngle0 = 90 -(-fAngleAircraft);
    printf("fAngle0: %.2f \n", fAngle0);
    fAdjacent = cos(fAngle0 * M_PI / 180) * (-fDistance);
    fOpposite = sin(fAngle0 * M_PI / 180) * (-fDistance);

    printf("fAdjacent: %.2f \n", fAdjacent);
    printf("fOpposite: %.2f \n", fOpposite);

    fNewY = fCurrentY + fAdjacent;
    fNewX = fCurrentX - fOpposite;
}

else if ((fAngleAircraft >= -180) && (fAngleAircraft < -90)) //Example 6
{
    fAngle0 = (-fAngleAircraft) - 90;
    printf("fAngle0: %.2f \n", fAngle0);
    fAdjacent = cos(fAngle0 * M_PI / 180) * (-fDistance);
    fOpposite = sin(fAngle0 * M_PI / 180) * (-fDistance);

    printf("fAdjacent: %.2f \n", fAdjacent);
    printf("fOpposite: %.2f \n", fOpposite);

    fNewY = fCurrentY + fAdjacent;
    fNewX = fCurrentX + fOpposite;
}
}

```

```

else if ((fAngleAircraft <= 180) && (fAngleAircraft > 90)) //Example 7
{
    fAngle0 = fAngleAircraft - 90;
    printf("fAngle0: %.2f \n", fAngle0);
    fAdjacent = cos(fAngle0 * M_PI / 180) * (-fDistance);
    fOpposite = sin(fAngle0 * M_PI / 180) * (-fDistance);

    printf("fAdjacent: %.2f \n", fAdjacent);
    printf("fOpposite: %.2f \n", fOpposite);

    fNewY = fCurrentY - fAdjacent;
    fNewX = fCurrentX + fOpposite;
}

else if ((fAngleAircraft <= 90) && (fAngleAircraft >= 0)) //Example 8
{
    fAngle0 = 90 - fAngleAircraft;
    printf("fAngle0: %.2f \n", fAngle0);
    fAdjacent = cos(fAngle0 * M_PI / 180) * (-fDistance);
    fOpposite = sin(fAngle0 * M_PI / 180) * (-fDistance);

    printf("fAdjacent: %.2f \n", fAdjacent);
    printf("fOpposite: %.2f \n", fOpposite);

    fNewY = fCurrentY - fAdjacent;
    fNewX = fCurrentX - fOpposite;
}

printf("New Y co-ordinate: %.2f \n", fNewY);
printf("New X co-ordinate: %.2f \n", fNewX);
printf("Therefore, (y;x) = ( %.2f ; %.2f ) \n", fNewY, fNewX);
printf("\n");
}

std::ostringstream ssCombined;

ssCombined << fNewY;
ssCombined << " ";
ssCombined << fNewX;

sTemp = boost::lexical_cast<std::string>(ssCombined.str());
printf("Combined ostream: %s \n", sTemp.c_str());

msgCoordinates.data = sTemp;

bAngle = false;
bDistance = false;
}

//Main method

```

```

int main(int argc, char **argv)
{
    //name assigned for the node
    ros::init(argc, argv, "boundingboxmove");

    ros::NodeHandle nh;

    //Topics that are subscribed to
    ros::Subscriber sub_distance = nh.subscribe("/DistanceFromBoundingBox", 10,
DistanceCallback);
    ros::Subscriber sub_angle = nh.subscribe("/AngleForLinearMovement", 10, AngleCallback);
    ros::Subscriber localpose_sub = nh.subscribe<geometry_msgs::PoseStamped>
        ("/mavros/local_position/pose",100,LocalposeCallback);

    //Topics that are published
    ros::Publisher coordinates_pub = nh.advertise<std_msgs::String>("/MultipleCoordinates",
10);

    ros::Rate r(10);

    printf("boundingboxmove.cpp has started...\n");

    while (ros::ok())
    {

        if ((bAngle == true) && (bDistance == true))
        {
            CoordinatesForAircraft();

            //Publishes the y and x coordinates for the aircraft to fly towards
            coordinates_pub.publish(msgCoordinates);
        }
        ros::spinOnce();
        r.sleep();
    }
}

```

Appendix 5.12: EulerToQuat.py node

```
#!/usr/bin/env python

#relevant libraries used in this node
import rospy
import math
from pyquaternion import Quaternion
from std_msgs.msg import String

roll = pitch = 0.0
yaw = 20

bReceived = 0
x = y = z = w = 0

#Callback to receive the angle from the bounding box obtained by using the darknet_ros package
def get_angle(msg):
    global bReceived
    global my_quat
    AngleReceived = msg.data
    print AngleReceived
# negative angle means aircraft must turn right
# positive angle means aircraft must turn left
# my_quat = Quaternion(axis=[0, 0, 1], angle=-0.349) #negative angle will rotate aircraft
right in gazebo
    AngleConverted = (float(AngleReceived))*math.pi/180
    print AngleConverted
#the angle is converted to a quaternion
    my_quat = Quaternion(axis=[0, 0, 1], angle=AngleConverted)
    bReceived = 1

#Name assigned for the node
rospy.init_node('EulerToQuat')

#Topic that is subscribed to
sub = rospy.Subscriber ('/AngleFromBoundingBox', String, get_angle)

#Topic that is published
pub = rospy.Publisher ('/QuatFromUser', String, queue_size=10)

print "EulerToQuat.py has started"

r = rospy.Rate(10)
while not rospy.is_shutdown():
    if (bReceived == 1):
        global my_quat
        # print my_quat
        (w,x,y,z) = my_quat
        tempstr = str(w) + " " + str(x) + " " + str(y) + " " + str(z)
```

```
    print tempstr
    bReceived = 0
#the quaternion is published to the topic QuatFromUser
    pub.publish(tempstr)
    r.sleep()
```

Appendix 5.13: QuatToEuler.py node

```
#!/usr/bin/env python

#relevant libraries used in this node
import rospy
import math
from geometry_msgs.msg import PoseStamped, Quaternion
from tf.transformations import euler_from_quaternion, quaternion_from_euler
from std_msgs.msg import String

#Declared variables

roll = pitch = 0.0
yaw = 20

bReceived = 0
x = y = z = w = 0

#callback to receive the quaternion from the aircraft
def get_quat(msg):
    global bReceived
    global CurrentQuat
    CurrentQuat = msg.pose.orientation
    #print CurrentQuat
    #print " "
    bReceived = 1;

#Name assigned for the node
rospy.init_node('QuatToEuler')

#Topic that is subscribed to
sub = rospy.Subscriber ('/mavros/local_position/pose', PoseStamped, get_quat)

#Topic that is published
pub = rospy.Publisher ('/AngleForLinearMovement', String, queue_size=10)

r = rospy.Rate(10)

print "QuatToEuler.py has started..."

while not rospy.is_shutdown():
    if (bReceived == 1):
        global CurrentQuat
        print CurrentQuat
        print " "
        RearrangeQuat = [CurrentQuat.x, CurrentQuat.y, CurrentQuat.z, CurrentQuat.w]
#the yaw is obtained by converting the quaternion to Euler angles
        (roll, pitch, yaw) = euler_from_quaternion(RearrangeQuat)
        YawDegrees = (float(yaw))*180/math.pi
```

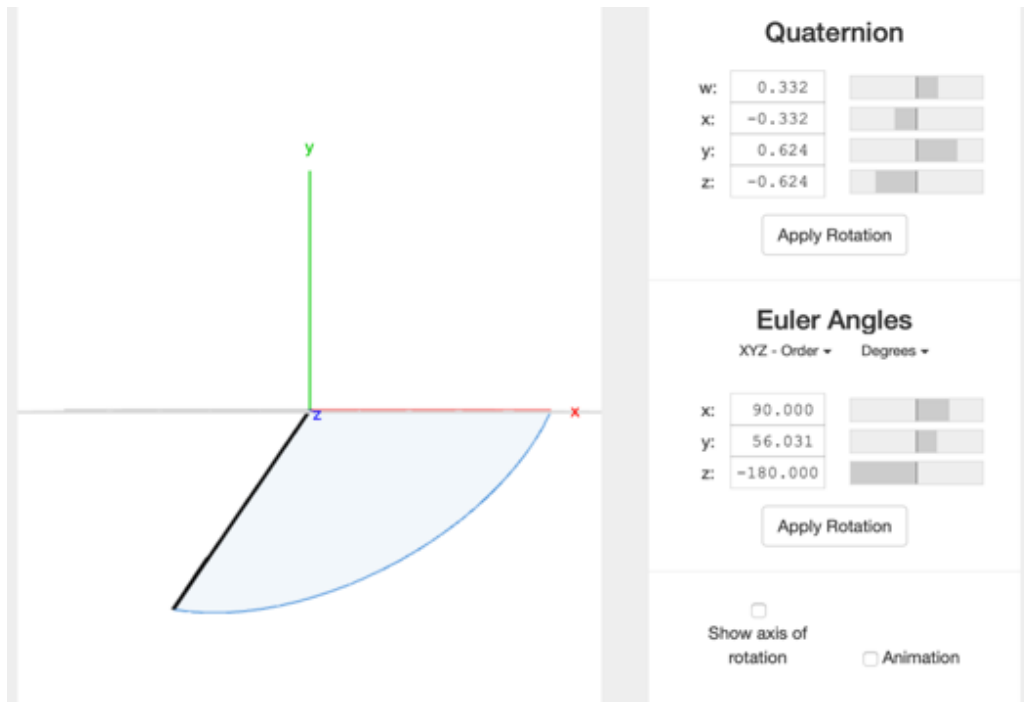
```
print YawDegrees
tempstr = str(YawDegrees)
print " "
bReceived = 0

#the yaw angle is published
pub.publish(tempstr)
r.sleep()
```

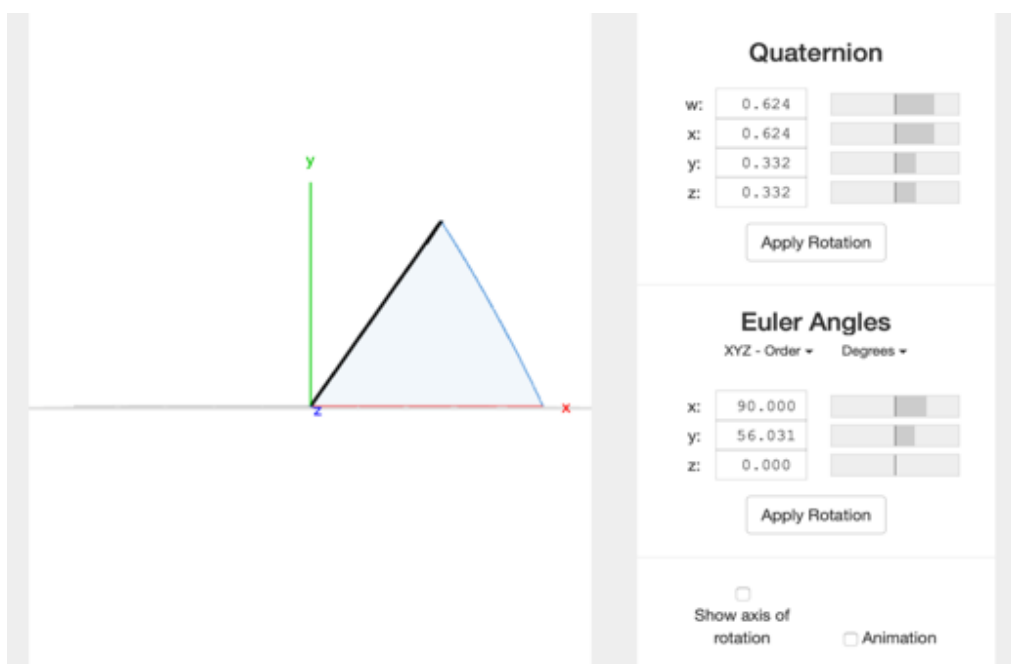

Appendix 5.14: Quaternion Simulator

All three screenshots were captured from the Quaternion Simulator (Quaternion Simulator, 2019).

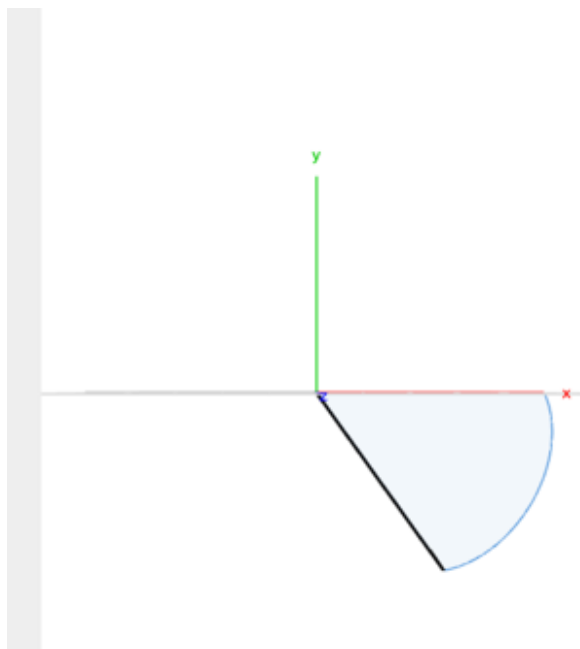
1) Convert the quaternion to Euler angles.



2) Subtract the z axis Euler angle by 180 degrees.



3) Invert the y axis Euler angle.



Quaternion

w:	0.624	<input type="text"/>
x:	0.624	<input type="text"/>
y:	-0.332	<input type="text"/>
z:	-0.332	<input type="text"/>

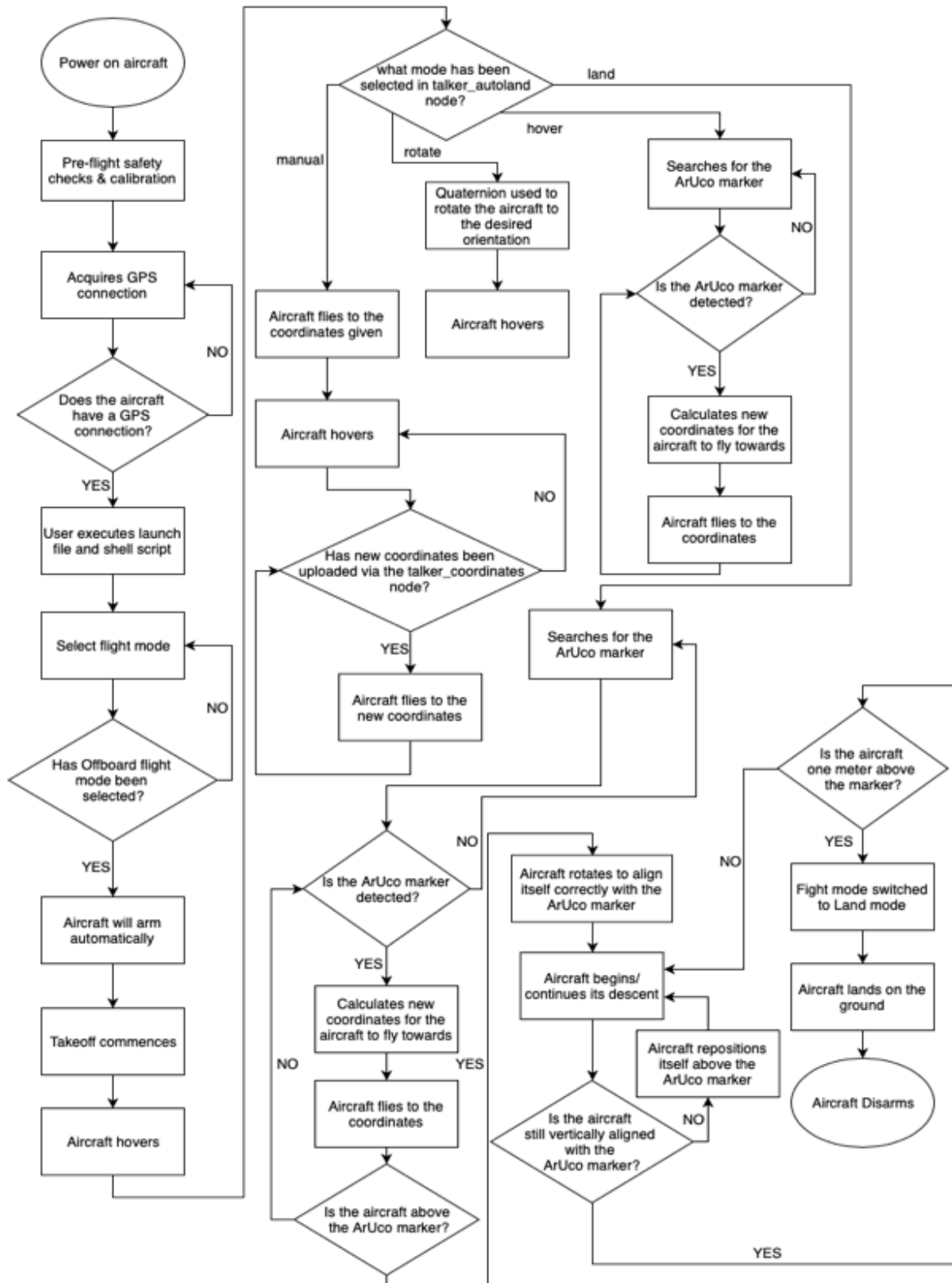
Euler Angles

XYZ - Order ▾ Degrees ▾

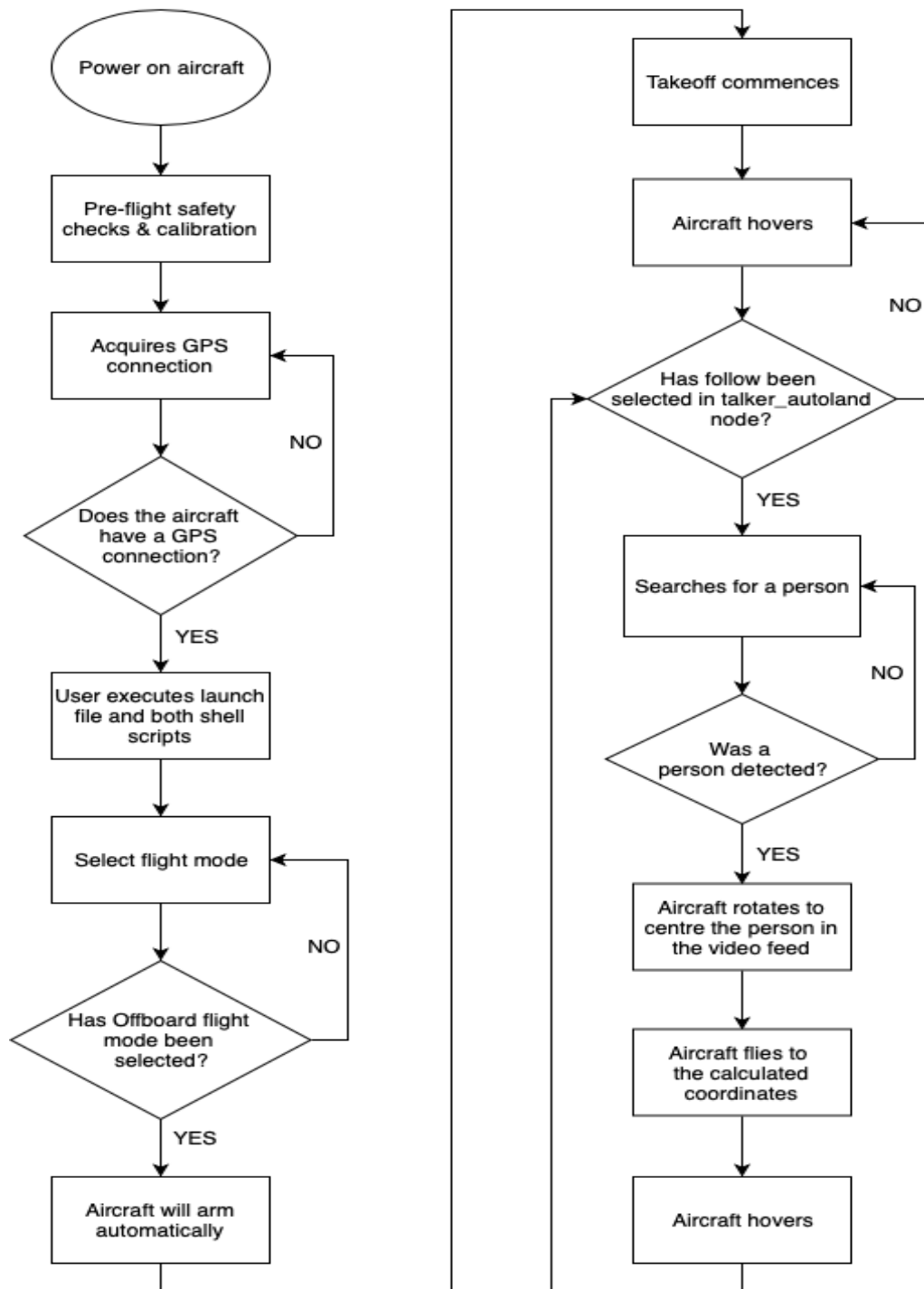
x:	90.000	<input type="text"/>
y:	-56.031	<input type="text"/>
z:	0.000	<input type="text"/>

Show axis of rotation Animation

Appendix 5.15: Developed Landing System Flow Chart



Appendix 5.16: Human Detection System Flow Chart



Appendix 6.1: Terminals Used for the Developed Landing System



Appendix 6.2: Additional Terminals Used for the Human Detection System

```
relab@relab-dell: ~  
┌─── /bin/bash 51x20 ───┐ ┌─── /home/relab/videostream_ws/src/video_stream_opencv/fa ───┐  
EulerToQuat.py has started  
-8.4987  
-0.148330297139  
0.997251025769 -0.0 -0.0 -0.0740971767462  
-12.1921  
-0.212792287732  
0.994345267669 -0.0 -0.0 -0.106195520923  
-16.4414  
-0.286956563637  
0.989724636882 -0.0 -0.0 -0.142986513869  
-18.7051  
-0.326465581915  
0.986707083094 -0.0 -0.0 -0.162508867978  
-18.5462  
-0.323692253733  
0.9869314  
-12.0332  
-0.210018  
0.994491569155 -0.0 -0.0 -0.10481659039  
└───┘ └───┘  
┌─── None 51x10 ───┐ ┌─── None 51x10 ───┐  
QuatToEuler.py has started...  
└───┘ └───┘  
┌─── /home/relab/darknet_ws/src/darknet_ros/darknet_ros/launcl ───┐ ┌─── None 51x10 ───┐  
FPS:0.9  
Objects:  
person: 73%  
└───┘ └───┘  
yavg: 436  
rotate right by 303 pixels  
rotate right by 12.03 degrees  
Distance between the aircraft and the person: 15.9 metres  
Distance aircraft needs to move: 12.10  
Angle and d  
└───┘ └───┘
```

EulerToQuat.py node

video_stream_opencv node

QuatToEuler.py node

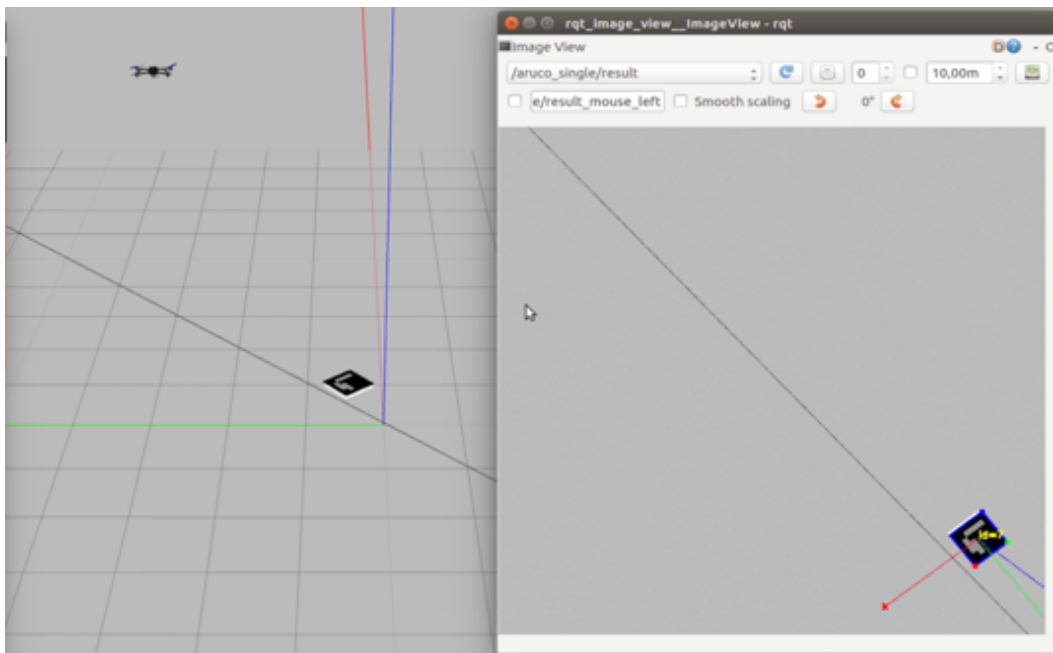
boundingboxmove.cpp

darknet_ros node

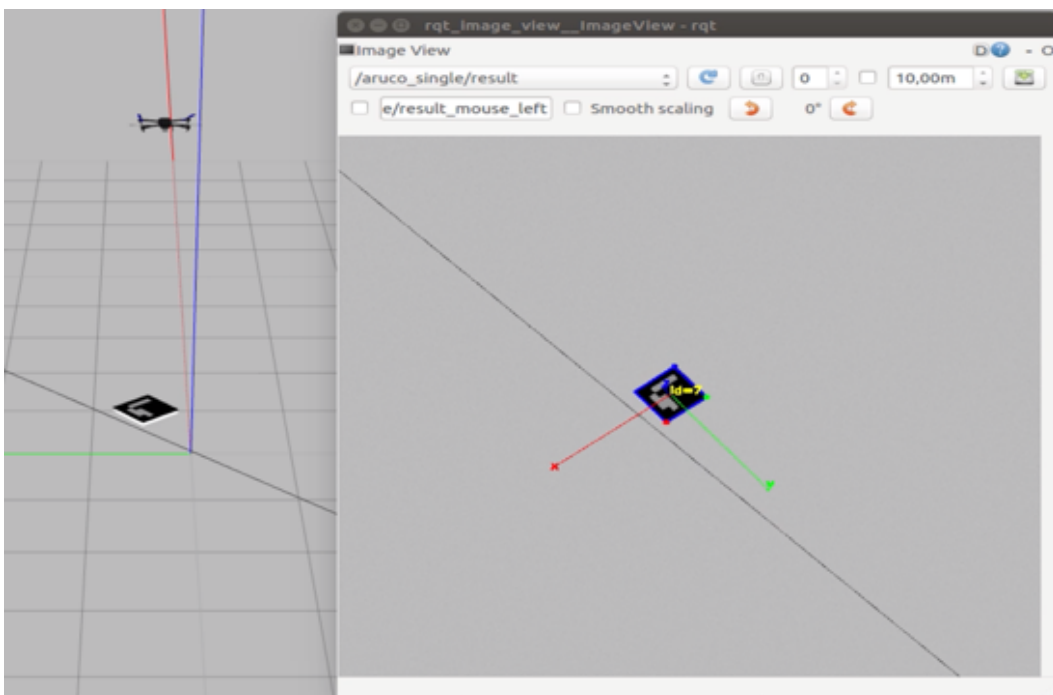
boxinfo.cpp node

Appendix 6.3: Testing of the Developed Landing System in Gazebo

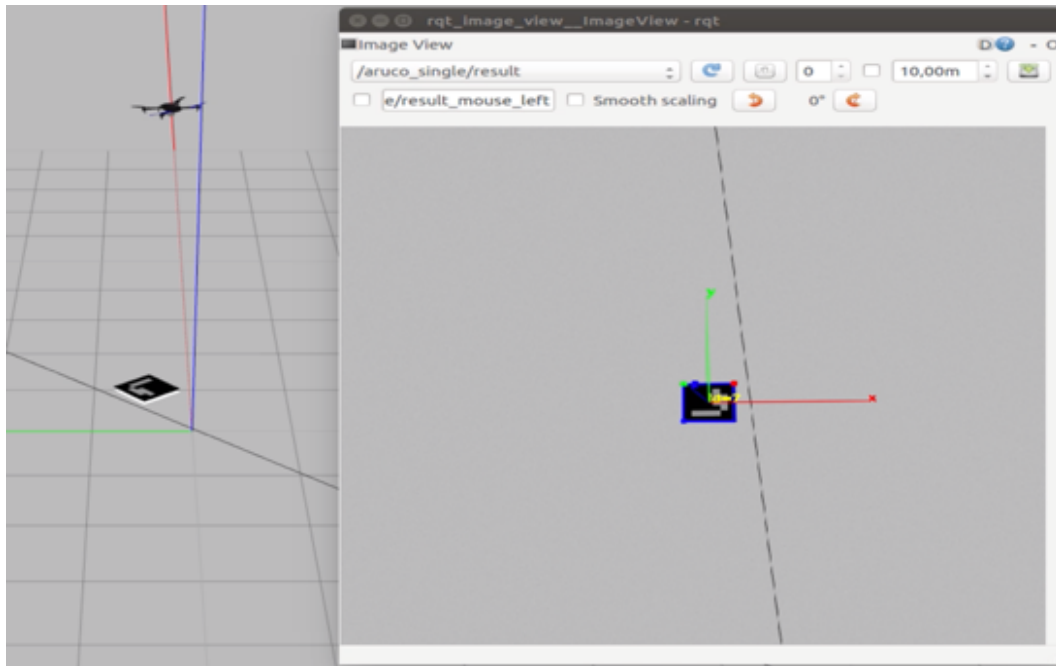
1. The position of the aircraft prior to executing the land command.



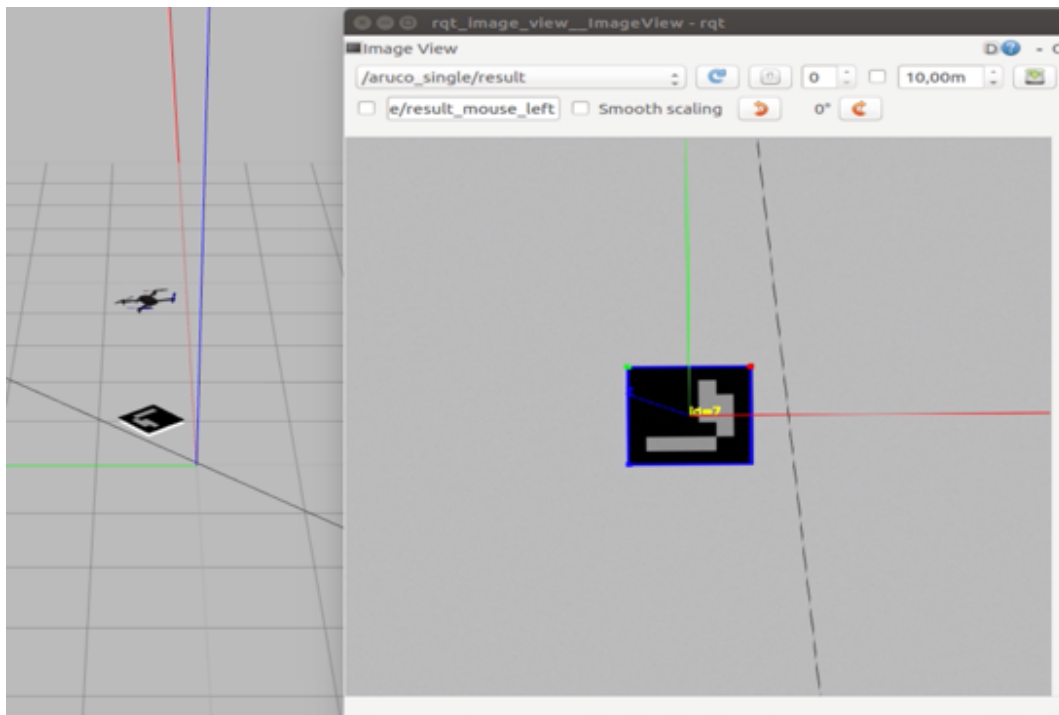
2. The aircraft repositioning itself above the ArUco marker and hovering in its position for a few seconds.



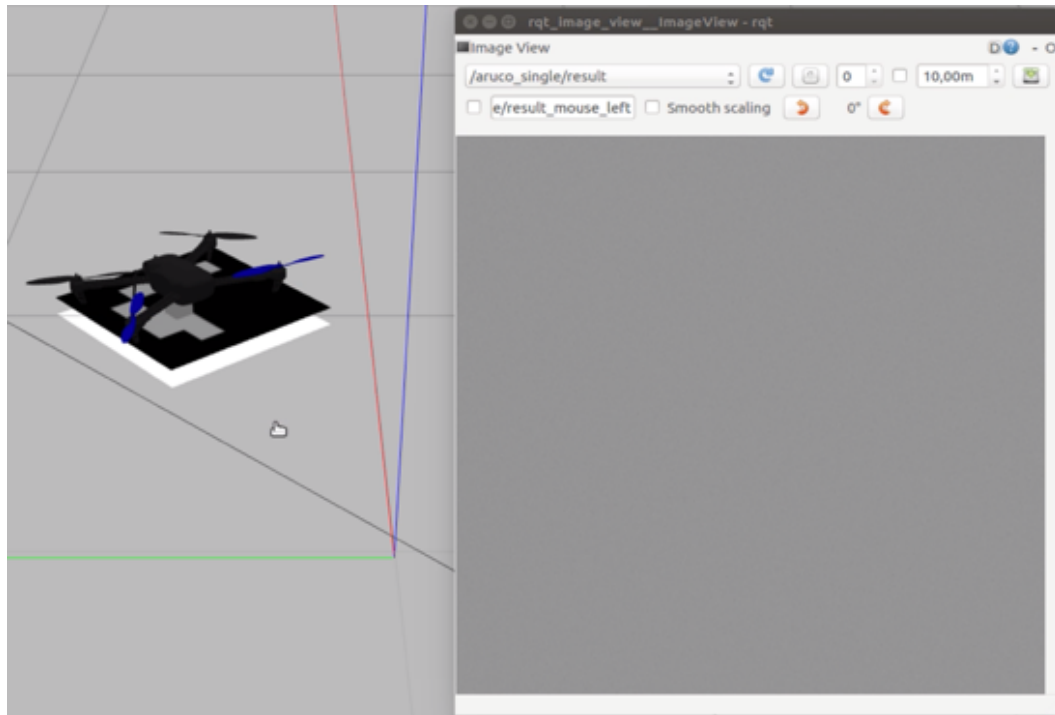
3. The aircraft rotating to align itself with the ArUco marker.



4. The aircraft starting its descent while constantly keeping itself aligned with the ArUco marker.



5. The aircraft landing correctly on the marker and disarming itself.



Appendix 7.1: Arduino PWM Script

```
//*****  
////////PPM Transmitter////////  
//*****  
  
//James Sewell  
//Benjy Nelson  
//MEng(Mechatronics)  
//Nelson Mandela University  
/*****/  
  
  
//Inital Variables and Configuration  
#define CHANNEL_NUMBER 4 //set the number of channels  
#define CHANNEL_DEFAULT_VALUE 1000 //set the default servo value  
#define FRAME_LENGTH 22500 //set the PPM frame length in microseconds (1ms = 1000µs)  
#define PULSE_LENGTH 500 //set the pulse length  
#define onState 1 //set polarity of the pulses: 1 is positive, 0 is negative  
#define sigPin 8 //set PPM signal output pin on the arduino  
  
int AUTO_PB = 2; //Green Auto Flight Mode Push Button Pin (Interrupt used)  
int ACRO_PB = 3; //White Acro (Manual) Flight Mode Push Button Pin (Interrupt used)  
int RTL_PB = 4; //Red RTL Flight Mode Push Button Pin (Interrupt used)  
int ppm[CHANNEL_NUMBER]; //ppm array  
  
int ButtonStateAuto;  
int ButtonStateAcro;  
int ButtonStateRTL;  
  
int LastButtonStateAuto = LOW;  
int LastButtonStateAcro = LOW;  
int LastButtonStateRTL = LOW;  
  
unsigned long LastDebounceTimeAuto = 0;  
unsigned long LastDebounceTimeAcro = 0;  
unsigned long LastDebounceTimeRTL = 0;  
  
unsigned long DebounceDelay = 50;  
  
int OutputStateAuto = LOW;  
int OutputStateAcro = LOW;  
int OutputStateRTL = LOW;  
  
int ButtonAuto = 0;  
int ButtonAcro = 0;  
int ButtonRTL = 0;  
  
unsigned long CurrentTimer = 0;  
int Temp = 0;  
  
bool AlreadyArmed = false;
```

```

void setup()
{
  pinMode(AUTO_PB, INPUT);
  pinMode(ACRO_PB, INPUT);
  pinMode(RTL_PB, INPUT);

  //Define push button interrupts
  attachInterrupt(5, AUTO_ISR, FALLING);
  attachInterrupt(4, ACRO_ISR, FALLING);
  attachInterrupt(3, RTL_ISR, FALLING);

  //initialize default ppm values
  for(int i=0; i<CHANNEL_NUMBER; i++)
  {
    ppm[i]= CHANNEL_DEFAULT_VALUE;
  }

  ppm[2] = 1300;

  pinMode(sigPin, OUTPUT);
  digitalWrite(sigPin, !onState); //set the PPM signal pin to the default state (off)

  cli();
  TCCR1A = 0; // set entire TCCR1 register to 0
  TCCR1B = 0;

  OCR1A = 100; // compare match register, change this
  TCCR1B |= (1 << WGM12); // turn on CTC mode
  TCCR1B |= (1 << CS11); // 8 prescaler: 0,5 microseconds at 16mhz
  TIMSK1 |= (1 << OCIE1A); // enable timer compare interrupt
  sei();

  Serial.begin(9600);
} //End of SETUP()

void loop()
{
  //Added Code
  ButtonAuto = digitalRead(AUTO_PB);
  ButtonAcro = digitalRead(ACRO_PB);
  ButtonRTL = digitalRead(RTL_PB);

  if (ButtonAuto != LastButtonStateAuto)
  {
    LastDebounceTimeAuto = millis();
  }

  if ((millis() - LastDebounceTimeAuto) > DebounceDelay)
  {

```

```

if (ButtonAuto != ButtonStateAuto)
{
  ButtonStateAuto = ButtonAuto;
  if (ButtonStateAuto == HIGH)
  {
//    OutputStateAcro = 0;
//    OutputStateRTL = 0;
//    OutputStateAuto = 1;

    ppm[0] = 1000; //Reset flightmode

    if (AlreadyArmed == false)
    {
      ppm[1] = 1000; //Set low PWM for throttle
      ppm[2] = 2000; //Set high PWM for yaw

      noInterrupts();

      CurrentTimer = millis();
      Serial.print("Before loop: ");
      Serial.println(CurrentTimer);
      int TempTime = (millis());
      while((millis() - CurrentTimer) < 5000)
      {

        Serial.print("Inside loop: ");
        Serial.print(CurrentTimer);
        Serial.print(" Temp Time: ");
        Serial.println(TempTime);
      }

      interrupts();
      ppm[1] = 1250;

      AlreadyArmed = true;
    }
    ppm[0] = 1300;
  }
}

if (ButtonAcro != LastButtonStateAcro)
{
  LastDebounceTimeAcro = millis();
}

if ((millis() - LastDebounceTimeAcro) > DebounceDelay)
{
  if (ButtonAcro != ButtonStateAcro)
  {

```

```

    ButtonStateAcro = ButtonAcro;
    if (ButtonStateAcro == HIGH)
    {
//    OutputStateAuto = 0;
//    OutputStateRTL = 0;
//    OutputStateAcro = 1;

        ppm[0] = 1600;
    }
}

if (ButtonRTL != LastButtonStateRTL)
{
    LastDebounceTimeRTL = millis();
}

if ((millis() - LastDebounceTimeRTL) > DebounceDelay)
{
    if (ButtonRTL != ButtonStateRTL)
    {
        ButtonStateRTL = ButtonRTL;
        if (ButtonStateRTL == HIGH)
        {
//            OutputStateAuto = 0;
//            OutputStateAcro = 0;
//            OutputStateRTL = 1;

                ppm[0] = 1900;
            }
        }
    }

/*
    Serial.print("Channel 1: ");
    Serial.print(ppm[0]);
    Serial.print(" Channel 2: ");
    Serial.print(ppm[1]);
    Serial.print(" Channel 3: ");
    Serial.println(ppm[2]);
*/

LastButtonStateAuto = ButtonAuto;
LastButtonStateAcro = ButtonAcro;
LastButtonStateRTL = ButtonRTL;
} //Irrelevant to programme but necessary for compile

//Auto push button ISR
void AUTO_ISR()
{

```



```

for(int i=0; i<CHANNEL_NUMBER; i++)
    ppm[i]= 1800;
}

//Acro push button ISR
void ACRO_ISR()
{
    for(int i=0; i<CHANNEL_NUMBER; i++)
        ppm[i]= 2000;
}

//RTL push button ISR
void RTL_ISR()
{
    for(int i=0; i<CHANNEL_NUMBER; i++)
        ppm[i]= 1000;
}

//Actual Output Sequence
ISR(TIMER1_COMPA_vect)
{ //leave this alone
    static boolean state = true;

    TCNT1 = 0;

    //Section A - High pulse section
    if (state)
    { //start pulse
        digitalWrite(sigPin, onState);
        OCR1A = PULSE_LENGTH * 2;
        state = false;
    }
    else
    { //End pulse and calculate when to start the next pulse
        static byte cur_chan_numb;
        static unsigned int calc_rest;

        digitalWrite(sigPin, !onState);
        state = true;
        //Section B - Low pulse for extended period of time
        if(cur_chan_numb >= CHANNEL_NUMBER) //Terminates packet transfer
        {
            cur_chan_numb = 0;
            calc_rest = calc_rest + PULSE_LENGTH;//
            OCR1A = (FRAME_LENGTH - calc_rest) * 2;
            calc_rest = 0;
        }
        //Section C - Low pulse for small period

```

```
else
{
OCR1A = (ppm[cur_chan_num] - PULSE_LENGTH) * 2;
calc_rest = calc_rest + ppm[cur_chan_num];
cur_chan_num++;
}
}
}
```

Appendix 7.2: InsertWaypoints.cpp node

```
//relevant libraries used in this node
#include <ros/ros.h>
#include <mavros_msgs/WaypointPush.h>
#include <mavros_msgs/CommandHome.h>
#include <mavros_msgs/WaypointClear.h>
#include <std_msgs/String.h>
#include <cstdlib>
#include <mavros_msgs/Waypoint.h>

//Declared variables
bool MissionClear = false;
bool MissionInsert = false;
bool SetHomePos = false;

//Main method
int main(int argc, char **argv)
{
    //name assigned for the node
    ros::init(argc, argv, "InsertWaypoints");
    ros::NodeHandle p;
    ros::NodeHandle n;
    ros::NodeHandle l;

    //Service Clients that are communicated with
    ros::ServiceClient wp_clear_client =
p.serviceClient<mavros_msgs::WaypointClear>("mavros/mission/clear");
    ros::ServiceClient wp_srv_client =
n.serviceClient<mavros_msgs::WaypointPush>("mavros/mission/push");
    ros::ServiceClient set_home_client =
l.serviceClient<mavros_msgs::CommandHome>("mavros/cmd/set_home");

    mavros_msgs::WaypointPush wp_push_srv;
    mavros_msgs::WaypointClear wp_clear_srv;
    mavros_msgs::CommandHome set_home_srv;

    mavros_msgs::Waypoint wp_msg;

    ros::Rate rate(20.0);

    ros::Time PrevTime = ros::Time::now();

    while(MissionClear != true)
    {
        if ((MissionClear != true) && (ros::Time::now() - PrevTime > ros::Duration(2.0)))
        {
            if (wp_clear_client.call(wp_clear_srv))
            {
                ROS_INFO("Waypoint list was cleared");
            }
        }
    }
}
```

```

        MissionClear = true;
    }
    PrevTime = ros::Time::now();
}

ros::spinOnce();
rate.sleep();
}

// wp_clear_srv.request = {};

set_home_srv.request.current_gps = false;
set_home_srv.request.latitude = -34.005196;
set_home_srv.request.longitude = 25.682365;
set_home_srv.request.altitude = 30.0;
PrevTime = ros::Time::now();

while(SetHomePos != true)
{
    if ((SetHomePos != true) && (ros::Time::now() - PrevTime > ros::Duration(2.0)))
    {
        if (set_home_client.call(set_home_srv))
        {
            ROS_INFO("Home was set to a new position");
            SetHomePos = true;
        }
        PrevTime = ros::Time::now();
    }

    ros::spinOnce();
    rate.sleep();
}

// MissionClear = false;

//A blank waypoint is uploaded
//////////////////////////////////// 0 //////////////////////////////////////
wp_msg.frame = 3;
// wp_msg.frame = 0;
wp_msg.command = 16;
wp_msg.is_current = false;
wp_msg.autocontinue = false;
wp_msg.param1 = 0;
wp_msg.param2 = 0;
wp_msg.param3 = 0;
wp_msg.param4 = 0;
wp_msg.x_lat = 1;
wp_msg.y_long = 2;
wp_msg.z_alt = 3;

```

```

wp_push_srv.request.start_index = 0;
PrevTime = ros::Time::now();
MissionInsert = false;

while(MissionInsert != true)
{
    if ((MissionInsert != true) && (ros::Time::now() - PrevTime > ros::Duration(2.0)))
    {
        wp_push_srv.request.waypoints.push_back(wp_msg);
        if (wp_srv_client.call(wp_push_srv))
        {
            // ROS_INFO("Waypoint list was cleared");
            ROS_INFO("Mission waypoints sent: %d", wp_push_srv.response.success);
//obtained from WaypointPush Service
            ROS_INFO("Waypoints sent: %d", wp_push_srv.response.wp_transferred);
            MissionInsert = true;
        }
        PrevTime = ros::Time::now();
    }

    ros::spinOnce();
    rate.sleep();
}

//The first waypoint is uploaded
//////////////////////////////// 1 //////////////////////////////////

wp_msg.frame = 3;
wp_msg.command = 22;
wp_msg.is_current = false;
wp_msg.autocontinue = false;
wp_msg.param1 = 0;
wp_msg.param2 = 0;
wp_msg.param3 = 0;
wp_msg.param4 = 0;
wp_msg.x_lat = -34.005196;
wp_msg.y_long = 25.682365;
wp_msg.z_alt = 30.0;

wp_push_srv.request.start_index = 0;
PrevTime = ros::Time::now();
MissionInsert = false;

while(MissionInsert != true)
{
    if ((MissionInsert != true) && (ros::Time::now() - PrevTime > ros::Duration(2.0)))
    {
        wp_push_srv.request.waypoints.push_back(wp_msg);
        if (wp_srv_client.call(wp_push_srv))
        {

```

```

        //          ROS_INFO("Waypoint list was cleared");
        ROS_INFO("Mission waypoints sent: %d", wp_push_srv.response.success);
//obtained from WaypointPush Service
        ROS_INFO("Waypoints sent: %d", wp_push_srv.response.wp_transferred);
        MissionInsert = true;
    }
    PrevTime = ros::Time::now();
}

    ros::spinOnce();
    rate.sleep();
}

//The second waypoint is uploaded
//////////////////////////////// 2 //////////////////////////////////
    wp_msg.frame = 3;
//    wp_msg.frame = 0;
    wp_msg.command = 16;
    wp_msg.is_current = false;
    wp_msg.autocontinue = false;
    wp_msg.param1 = 0;
    wp_msg.param2 = 0;
    wp_msg.param3 = 0;
    wp_msg.param4 = 0;
    wp_msg.x_lat = -34.005310;
    wp_msg.y_long = 25.682194;
    wp_msg.z_alt = 30.0;

    wp_push_srv.request.start_index = 0;
    PrevTime = ros::Time::now();
    MissionInsert = false;

    while(MissionInsert != true)
    {
        if ((MissionInsert != true) && (ros::Time::now() - PrevTime > ros::Duration(2.0)))
        {
            wp_push_srv.request.waypoints.push_back(wp_msg);
            if (wp_srv_client.call(wp_push_srv))
            {
                //          ROS_INFO("Waypoint list was cleared");
                ROS_INFO("Mission waypoints sent: %d", wp_push_srv.response.success);
//obtained from WaypointPush Service
                ROS_INFO("Waypoints sent: %d", wp_push_srv.response.wp_transferred);
                MissionInsert = true;
            }
            PrevTime = ros::Time::now();
        }

        ros::spinOnce();
        rate.sleep();
    }
}

```

```

//The third waypoint is uploaded
//////////////////////////////////// 3 //////////////////////////////////////
    wp_msg.frame = 3;
//    wp_msg.frame = 0;
    wp_msg.command = 16;
    wp_msg.is_current = false;
    wp_msg.autocontinue = false;
    wp_msg.param1 = 0;
    wp_msg.param2 = 0;
    wp_msg.param3 = 0;
    wp_msg.param4 = 0;
    wp_msg.x_lat = -34.005329;
    wp_msg.y_long = 25.683224;
    wp_msg.z_alt = 30.0;

    wp_push_srv.request.start_index = 0;
    PrevTime = ros::Time::now();
    MissionInsert = false;

    while(MissionInsert != true)
    {
        if ((MissionInsert != true) && (ros::Time::now() - PrevTime > ros::Duration(2.0)))
        {
            wp_push_srv.request.waypoints.push_back(wp_msg);
            if (wp_srv_client.call(wp_push_srv))
            {
                //                ROS_INFO("Waypoint list was cleared");
                ROS_INFO("Mission waypoints sent: %d", wp_push_srv.response.success);
//obtained from WaypointPush Service
                ROS_INFO("Waypoints sent: %d", wp_push_srv.response.wp_transferred);
                MissionInsert = true;
            }
            PrevTime = ros::Time::now();
        }

        ros::spinOnce();
        rate.sleep();
    }

    ros::spinOnce();

    return 0;
}

```

Appendix 7.3: Testing of the Developed Landing System on the Actual Aircraft

1. The aircraft manually positioned using the `talker_coordinates.cpp` node, where “land” mode will be activated using the `talker_autoland.cpp` node.



2. The aircraft repositioned above the ArUco marker.



3. The aircraft rotated to align itself correctly with the ArUco marker.
(Take note of the direction that the tail of the aircraft is facing).



4. The aircraft descending towards the ground.



5. The aircraft landed successfully on the ArUco marker.



Appendix 7.4: Tables for Test 1, Test 2 and Test 3 for the Human Detection System

Test 1					
Position	Xmin	Ymin	Xmax	Ymax	Xaverage
home	null	null	null	581	960
1	1179	444	1237	586	1208
A	920	437	971	577	945.5
2	1414	498	1482	631	1448
B	917	454	959	577	938
3	1701	449	1790	587	1745.5
C	953	396	995	522	974
4	1793	481	1885	611	1839
D	939	374	973	498	956

Test 2					
Position	Xmin	Ymin	Xmax	Ymax	Xaverage
home	null	null	null	931	960
1	1292	772	1359	920	1325.5
A	915	693	969	837	942
2	1532	725	1626	860	1579
B	909	628	958	756	933.5
3	1809	699	1913	841	1861
C	957	510	1010	637	983.5

Test 3					
Position	Xmin	Ymin	Xmax	Ymax	Xaverage
home	null	null	null	385	960
1	1175	269	1226	385	1200.5
A	952	255	1001	375	976.5
2	1326	240	1379	358	1352.5
B	988	217	1032	337	1010
3	1525	239	1586	353	1555.5
C	935	182	971	295	953
4	1714	207	1782	330	1748
D	945	141	981	250	963