De-Mining with UAVs

A Major Qualifying Project

Submitted to the faculty of
WORCESTER POLYTECHNIC INSTITUTE
In partial fulfillment of the requirements for the
Degree of Bachelor of Science

Submitted By
Arthur Lockman
Greg Tighe
Tucker Haydon

Submitted To
Craig Putnam
William Michalson

Abstract

The goal of this project is to use a low cost Unmanned Aerial Vehicle (UAV) to drop cheap and readily available payloads onto landmines to detonate them, specifically targeting the PMN-1 antipersonnel landmine. This project assumes that the landmines have been marked by another robot. This system is designed to be much cheaper than current solutions so as to be affordable to civilians. The project produced a set of autonomous routines that can be used to deploy mine-destroying payloads, as well as a payload deployment system and a set of recommendations that can be used to produce a fully-functional prototype of this system.

Contents

Abstract **ii**

1 Introduction **1**

2 Background **3**

2.1 Worldwide Impact of Landmines 3

2.1.1 History of Development 3

2.1.2 Ban on Landmines 4

2.2 The PMN-1 Landmine 5

2.3 Current Landmine Removal Techniques 7

2.4 Proposed Solution 9

2.4.1 Cost 10

2.4.2 Availability of Components 11

2.4.3 Advantages 11

2.4.4 Disadvantages 11

2.4.5 Constraints 12

2.4.6 Challenges 13

3 Methodology & Experimentation **14**

3.1 Testing Device 14

3.1.1 Design and Construction 14
3.1.2 Calibration and Code Issues 16
3.2 Testing 17
3.3 Shrapnel Calculations 18
3.4 UAS Selection 20
3.4.1 Airframe 20
3.4.2 Flight Controller 21
3.5 UAS Setup 21
3.5.1 Frame 22
3.5.2 Wiring 24
3.5.3 Controller and Radios 24
3.5.4 Ground Control Software 25
3.5.5 Failsafes 26
3.5.6 Power and Battery 26
3.6 Payload Development and Testing 26
3.7 Payload Bay Design 28
3.8 Vision System 30
3.8.1 Lens Selection 30
3.8.2 Vision Target Choice 31
3.8.3 Algorithm Modifications 32
3.8.4 Target Identification Algorithm 33
3.8.5 Optimization 35
3.8.6 Image Streaming 36
3.9 Drone Electronics 38
3.10 System Integration 40

3.10.1 System Organization 40
3.10.2 Main Control Loop 42
3.11 Drone Control Interface 43
3.12 Centering and Deployment 44

4 Results 46

4.1 Flight Control System 47
4.2 Target Identification System 47
4.3 Payload Delivery System 47
4.4 GPS Data System 48

5 Discussion 49

5.1 Airframe 49
5.2 Payload Bay Prototypes 50
5.3 Drone Control Interface 50
5.4 DroneKit & Mavlink Interfaces 51

6 Conclusion 53

7 Future Work 54

7.1 Mine Detection Robot 54
7.2 Mine Detonation Verification 54
7.3 Further Target Development 55
7.4 Centering Algorithm 55
7.5 Payload Bay 56

References 57

Appendix A System Flow Diagrams **58**

Appendix B Source Code **62**

B.1 Python Image Streaming Code 63

B.2 MATLAB Code 64

B.3 Data Simulator 68

B.4 Shell Code 70

B.5 DroneKit Interface Code 71

B.6 Target Identification Code 89

B.7 GPS Code 96

Appendix C UAS Operation and Maintenance Procedure **98**

Appendix D Flight Safety **100**

D.1 Flight 100

D.1.1 Preparation 100

D.1.2 Takeoff 101

D.1.3 Flying Safety 101

D.1.4 Breakdown 102

D.1.5 Crash Landing 102

D.2 Batteries 102

D.2.1 Storage 102

D.2.2 Charging 102

D.2.3 Safety 103

D.2.4 Physical signs of damage: Swelling 103

D.2.5 Electrical signs of damage and electrical safety: 104

D.3 Props 104

List of Figures

2.1 The PMN-1 mine (left) pictured next to its newer cousin, the PMN-2 (*PMN-Mine*, n.d.-a). 5

2.2 PMN-1 Landmines found in Iraq, 2003 (*PMN Mine*, n.d.-b). 6

2.3 Cross-section of the PMN-1 land mine (*Soviet / Russia PMN-1 Bakelite Landmine*, n.d.). 6

2.4 Common flail mine removal device (*Mine flail*, n.d.). 7

2.5 Truck-mounted land mine roller device (*Landmine and IED Rollers - HRI*, n.d.). 8

2.6 Front view of the chosen airframe, wings extended. 9

3.1 Internal configuration of strain gauge and plates. 15

3.2 The practice mine. 15

3.3 Strain gauge and filter circuit diagram. 16

3.4 Calibration curve for testing device 17

3.5 Payload drop data for drop on testing mine (dropped from 40in) 18

3.6 Shrapnel calculation graphs for mean steel band shrapnel shards. 19

3.7 Shrapnel calculation plots for distance and velocity vs time. 19

3.8 Side view of airframe, wings extended. 22

3.9 Top view of airframe, wings extended. 23

3.10 Airframe with wings folded down. 23

3.11 Directions and connection information for an OctoCopter using a Pixhawk, from http://copter.ardupilot.com/wiki/connect-escs-and-motors/ 24

3.12 Annotated view of controller, showing function each control is mapped to. . 25

3.13 Water bottle based payload design. 27

3.14 Final payload design, based on plastic wrap filled with sand. 27

3.15 Original payload bay design. 28

3.16 Payload bay trapdoor and mounting plate. 29

3.17 Final payload bay design, showing both the CAD model and the finished product mounted on the UAV. 29

3.18 Target development progression, with the oldest on the left and the final on the right. 32

3.19 The target identification algorithm 34

3.20 Image Processing Steps 34

3.21 Image capture routine. 36

3.22 Vision targets. 37

3.23 Top of the airframe, showing the layout of the various controller components. 39

3.24 H-bridge chip board for controlling the solenoids. 40

3.25 Overall system data and control flow. 41

3.26 Main control loop program flow. 42

A.1 Check image processing queue flow diagram. 58

A.2 Shell command processing flow diagram. 59

A.3 Image display process flow diagram. 60

A.4 Old target identification algorithm process diagram. 60

A.5 Target identification algorithm process diagram. 61

A.6 Manual override check process flow diagram. 61

D.1 Battery safely charging in charging bucket. 103

List of Source Code

B.1 Python image streaming code 63
B.2 Shrapnel maximum distance plotting code 64
B.3 Shrapnel maximum velocity plotting code 66
B.4 Data simulator code 68
B.5 Shell code 70
B.6 Flight control interface code 71
B.7 Target identification code 89
B.8 GPS input code 96

Chapter 1

Introduction

This project is designed to create a safe, reliable, and relatively cheap method for the disposal of PMN-1 and similar class landmines. To maximize safety, the system uses an Unmanned Aerial System (UAS) to detonate the landmines by dropping a payload upon them, removing the need for humans to even approach and interact with a minefield. The system is designed to make use of existing technology to keep costs down. Low costs allow the system to be purchased by de-mining companies or small towns that need a cheaper, safer solution for de-mining. This is not a mine detection project; this project assumes knowledge of landmines identified and marked by a previous robot. This project also assumes that it will not be used in live combat, meaning that the de-mining task is not extremely time sensitive.

Demining is a challenge that impacts many people's lives across the globe. Thousands of active cold war era landmines (such as the PMN landmines) are still buried in countries like Iraq, Afghanistan, Sudan, Syria and Cambodia. Many of these mines explode daily, taking the feet, legs or lives of noncombatants removed from war. This lack of target discrimination is what led to the near global ban on landmines in 1997, though landmines are still seemingly being planted in war zones today. To prevent the further mutilation of noncombatants, better techniques for both detection and disposal of landmines need to be developed.

This project will assist in the removal of cold-war era PMN anti-personnel landmines. This class of landmines is one of the most widely used landmines throughout the world due to its low production cost and simple design. These mines are still found in Iraq and new reports suggest that they are being deployed in Syria. As a result of its simple design and widespread use, it is the ideal target for a useful and effective remote detonation system. A low-cost disposal system could help thousands of people around the globe and truly make a difference.

In recent years multirotor UAS's have dropped significantly in price and have become widely available, making them an ideal platform for this project. An autonomous multirotor equipped with multiple payloads, target identification, and reload algorithms could potentially clear a minefield very quickly, making it a valuable asset to governments, cities, contractors, or other companies.

Chapter 2

Background

This section details the background research completed for this project. This includes a summary of the current need for land mine removal tools, an analysis of current technologies available for demining, and a description of the proposed solution for this project.

2.1 Worldwide Impact of Landmines

Each year thousands of peoples' lives are forever changed by landmines and other unexploded munitions (Monitoring & Committee, 2014). These munitions lie hidden in the ground for decades, distributed throughout the world, waiting for an unknowing victim to trigger them. In a moment, dozens of peoples' lives change. Parents lose children, children lose parents, husbands lose wives and vice versa. People lose the ability to walk, to work, to communicate, or to support a family. These munitions are remnants of wars generations in the past but still affect the lives of many people today.

2.1.1 History of Development

The development of landmines predates the US Civil War, but the modern-day mechanically-activated landmine was developed during the end of the war (circa 1864). In the following 50 years, warfare was historically light and the technology was not developed any further until the breakout of WWI. WWI brought upon the industrial age of warfare. Planes, tanks, trenches, submarines and other new technologies were developed to lead the assaults, so defensive measures were created to counter these. To counter the British introduction of tanks, the Germans developed the first anti-tank mines which then developed quickly into anti-personnel mines by the end of 1918 (Youngblood IV, 2002).

In the interim years between WWI and WWII, militaries realized the defensive value of landmines and quickly sought to develop the technology further. During WWII, the Germans alone created over fifty different types of anti-tank and anti-personnel landmines. Alongside the development of compact explosives, shrapnel, and detonation triggers came a change in the purpose of mines. Anti-personnel landmines quickly developed into a mental-warfare tool as much as an explosive-warfare tool. Anti-personnel mines were soon designed to cause injuries, not fatalities. Militaries realized that it was much more costly to recover and rehabilitate injured soldiers than it was to recover cadavers with the added benefit of a severe decrease in solider moral (Youngblood IV, 2002).

After their widespread success in WWII, landmines became standard in many military conflicts. They were used many times in the proxy wars fought between the US, USSR, and China during the Cold War. Because of their low cost and difficulty to be collected and reused, landmines were often airdropped over enemy territory in an effort to disrupt supply lines and deny advances. During the Vietnam War, the US employed this technique extensively to disrupt the Ho Chi Minh Trail, resulting in many square kilometers of 'danger zones' (Youngblood IV, 2002). These danger zones persist for decades after the end of the wars, often lost and forgotten before being rediscovered by a civilian or child. The mounting civilian casualties from leftover mine fields resulted in an international effort to ban the use of landmines in 1997 (Monitoring & Committee, 2014).

2.1.2 Ban on Landmines

In 1997, the Mine Ban Treaty was signed by 122 countries. The treaty promised the end of landmine production, stockpiling, and development, a destruction of current stockpiles, and an investment in demining technologies. Currently there are 162 signatories excluding namely the United States, China, Russia (Monitoring & Committee, 2014).

The push for the ban on landmines is driven by one major problem: an absurdly high rate of civilian casualties. In 2013, 79% of the 3,308 landmine victims were civilians, of whom nearly half (1,300) were children. The problem with landmines is that they are a non-discriminatory weapon. They do not differentiate between the footstep of a child and the footstep of a soldier. In addition, fields can persist for decades, and there is no real way of detecting a landmine before somebody steps on one. They are traps waiting to take the life of unknowing civilians (Monitoring & Committee, 2014).

The ban on landmine production, efforts in active demining, and increased awareness of landmine dangers has greatly reduced the number of worldwide casualties. Since 1999, the number of landmine casualties has decreased from 9,220 to 3,308. Since 2008, over 200 square kilometers of mined area has been cleared. In 2013 alone, nearly 300,000 mines were destroyed. Despite these efforts, there are still many 'heavily mined' areas. The 2014 Landmine Monitor reports that more than 100 square kilometers of land is 'massively mined' in Afghanistan, Cambodia, Turkey and Iraq. Anywhere between 20 and 100 square kilometers of land is 'heavily mined' in Angola, Azerbaijan, Croatia, Thailand, and Zimbabwe. The international ban has greatly helped the world in reducing landmine contamination but there is still a massive amount of work to be done (Monitoring & Committee, 2014).

2.2 The PMN-1 Landmine

Developed by the Soviet Union in the late 1960's and cloned extensively by China and Hungary, the PMN type land mine is one of the most widely used landmines. The mine itself can be seen in Figure 2.1. Its cheap design, high explosive content, and light trigger pressure makes it one of the most cost effective–yet volatile–landmines to date (Swinton & Bergeron, 2004). To give an idea of the production extent of this mine, Ukraine alone inherited 404,903 PMN type landmines after the dissolution of the Soviet Union (Group, 2004). Millions of other PMN mines were distributed among the other dissolved states. These mines are used extensively throughout the Middle East and Southeast Asia as demonstrated by Figure 2.2 below.

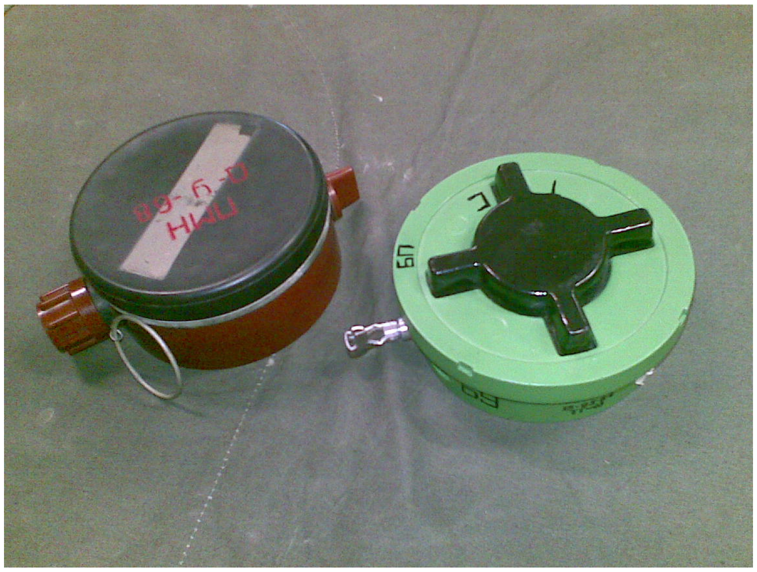


Figure 2.1: The PMN-1 mine (left) pictured next to its newer cousin, the PMN-2 (*PMN-Mine*, n.d.-a).

The trigger force of the PMN-1 landmine is reported as ranging from 10 - 100N, though most sources report it as about 50 newtons (*Soviet / Russia PMN-1 Bakelite Landmine*, n.d.). Generally the purpose of a landmine is not to kill a target, rather severely maim them, but the PMN mine is different. When compared to other landmines, the PMN-1 is particularly deadly containing 240g TNT as compared to the 30-40g TNT most other landmines have (*Soviet / Russia PMN-1 Bakelite Landmine*, n.d.).

Figure 2.2: PMN-1 Landmines found in Iraq, 2003 (*PMN Mine*, n.d.-b).

The mine itself is very simple in construction (cross-section shown in 2.3. The top of the land mine consists of a Bakelite sheet covered in a thin rubber. This rests on top of the firing pin of the mine. When a foot or other weight presses on the top of the mine, it depresses the firing pin which causes the mine to explode, causing shrapnel of Bakelite, rubber, and steel to fly out.

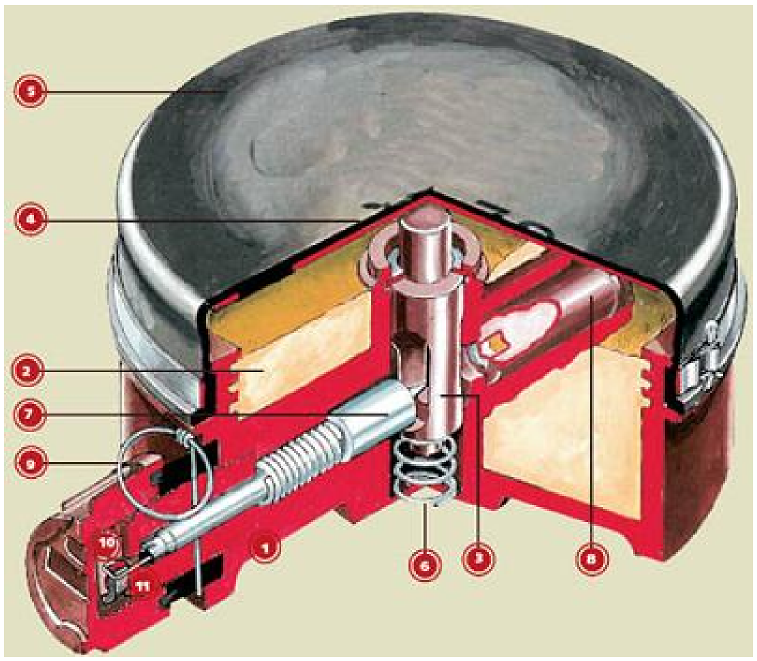


Figure 2.3: Cross-section of the PMN-1 land mine (*Soviet / Russia PMN-1 Bakelite Landmine*, n.d.).

2.3 Current Landmine Removal Techniques

There are already several solutions to destroying landmines, and the only ones available to civilian groups are large armored vehicles that are costly and require special training to use. There is not much numerical data on how effective these tools are, but empirical evidence would suggest that these mechanical systems are fairly effective.

Flail type vehicles were originally created around World War 1 and use long chains on a rotating drum to detonate or destroy mines. The chains can have weights on the ends to create more force when they hit the ground. The idea is that if one of these chains or weights hits a land mine it will act as though someone had stepped on the mine and the mine will detonate. The weights also send shockwaves through the ground on impact, which can trigger more sensitive mines. A large shield sits between the machine and the flail to protect the driver and machine from any kind of explosion. An example of a flail mine removal device can be seen in Figure 2.4.

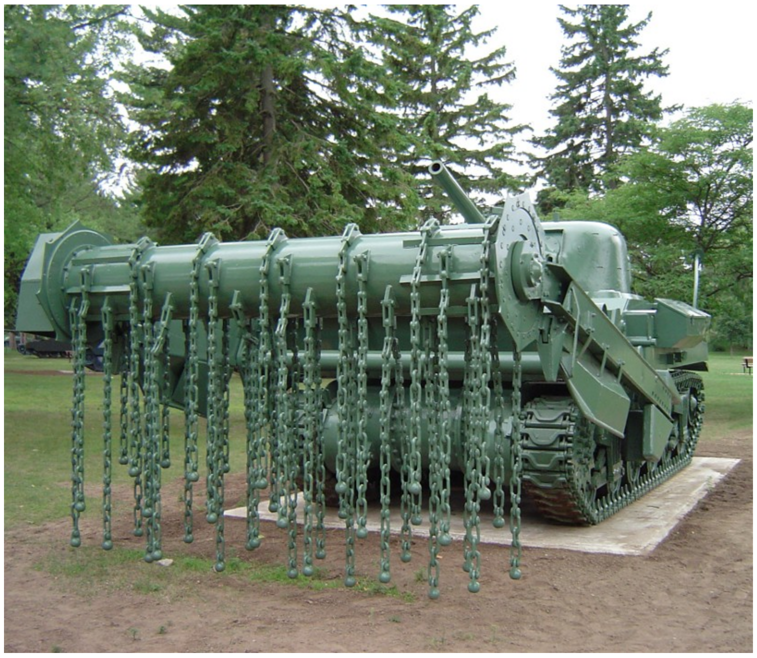


Figure 2.4: Common flail mine removal device (*Mine flail*, n.d.).

Till type vehicles use a large spinning toothed wheel or plow that cuts through the dirt to unearth and destroy mines. Again, a large shield sits between the till and machine to protect the driver and machine from explosions.

Both flails and tills have similar issues as they operate along the same basic principle of a spinning tool with lots of small attachments to destroy the mines. The first type of issue is called a disruptive strike, and this involves a tool hitting a mine and damaging it, but not detonating it. In this case the mine may either be so damaged that it cannot function, or it may become even more dangerous. It also can spread shrapnel from an unexploded mine over a large area which can cause issues with metal detector sweeps, as each time the detector finds something the metal must be found before the sweeper can move on. The other issue that both methods have in common is the possibility for ejection, in which an unexploded mine is removed from the ground and thrown clear of the machine. While rare, this has the chance to spread mines into areas that have already been cleared and cause teams to have to do a second sweep to clear the area again. It also has the obvious danger of flying explosive devices. Flails specifically have issues where sometimes there will be streaks of unbeaten areas because of the way that the chains have been laid out on the drum or missing chains due to explosions. While this is less likely for a till, there have been reports of small waves of dirt riding the front of the till and carrying mines for a large distance.

Rollers are exactly what they sound like: a large metal roller that is pushed in front a vehicle to destroy mines. Most of these are made on site with available materials and are not always very effective. Some rollers like the one pictured in Figure 2.5 are much larger and more effective than hand-made rollers. These are often mounted on the front of trucks and driven over minefields to clear them.

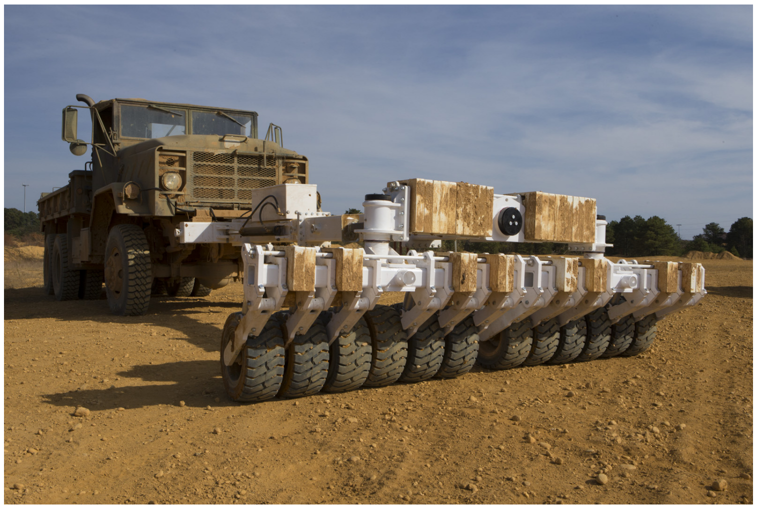


Figure 2.5: Truck-mounted land mine roller device (*Landmine and IED Rollers - HRI*, n.d.).

The last widely used method is to excavate the dirt and process it. Some machines do everything on board and sift for solid matter while immediately replacing the dirt excavated with sifted soil. Other systems will remove the dirt and lay it out for inspection or run it through toughened rock crushers to destroy the mines.

All of these systems share a few issues, the biggest of which is terrain. If the terrain is too rocky or rough the systems cannot function properly. Another issue is with soil quality of the location, as hard packed dirt acts very differently than a softer topsoil or sand. The last of the issues is that these machines can cause a lot of dust or debris to be thrown up and can obstruct the drivers view, which can lead to uneven clearing of a field. While these systems do put operators near danger they do protect the operator with armored cabins, and some vehicles are even remote controlled to further protect the operator.

Most of these systems are available for either renting or purchase. The costs are not publicly listed, but these solutions require additional training, as well as paid mechanics. The costs quickly add up. Commonly, these solutions can be rented with a support crew, but this still costs on the order of multiple thousands of dollars.

2.4 Proposed Solution

The proposed demining solution is to use an autonomous UAS to identify marked landmines and detonate them by dropping a mass on them.

For this project, two options for the UAS were considered: either a fully-built kit model or a personally custom-designed and built model using off the shelf parts. A fully-built kit would be the most reliable but a custom-build kit would be widely-available for anyone to build. A goal of this project was to use components that would be easily available such that the system could be easily reproduced, so ideally the custom-build UAS would have been chosen. In reality, the project required a significant amount of precision that could not be 100% guaranteed by a custom UAS within the allotted time frame. Thus, a fully-built UAS was chosen as the payload delivery platform. The selection of the specific airframe is discussed further in section 4.4.

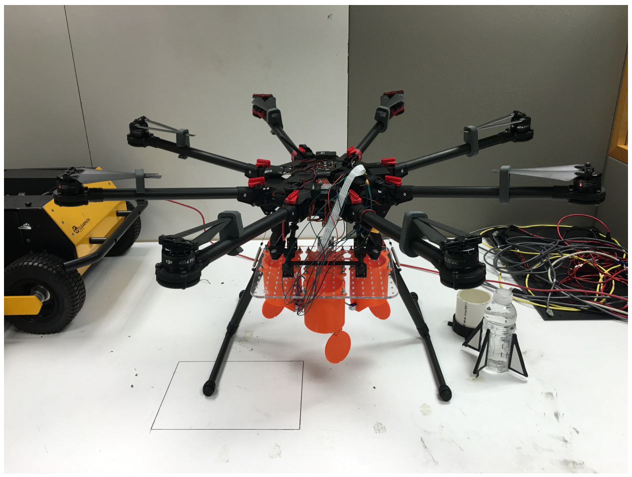


Figure 2.6: Front view of the chosen airframe, wings extended.

The proposed system is semi-autonomous, requiring minimal operator training. The detected mines are marked by paper or cloth squares such that the UAS detects them using a camera.

Once detected, the proposed system will center above a landmine target and drop a payload upon the mine, detonating it. The UAS accelerates away from the mine immediately after releasing the payload, avoiding any shrapnel. Visual or audio confirmation will confirm whether or not a landmine was detonated.

2.4.1 Cost

The project will cost somewhere in the range of \$4,000 to \$5,000. This includes the cost of the vision system, the UAS, the testing equipment and some extra parts. Estimated system costs are given in Table 2.1.

The drone selected for this project will need to have a few key components. It will need to be able to lift a cargo capacity of 4-6kg in order to accommodate carrying the payload deployment system and a few payloads, as well as the camera system. The drone itself will need to be equipped with a programmable flight controller, as well as a GPS and programmable ground control radio system. These constraints allow for the selection of either a pre-built UAS with the correct features, or the construction of a custom one to fit the requirements exactly.

The drone camera system that is selected will need to be able to interface with the flight controller and transmit camera data back to the ground control computer. The camera will also need to be able to do the required pre-placed flag identification in flight without waiting for a response from the ground control computer. The camera will also need to be able to identify moving targets when flying at a minimum of 20 feet in the air, which is the desired drop altitude for the payloads.

The testing system itself will be constructed from standard hardware-store PVC pipe, rubber sheet, and plywood. The electronics inside it will be a strain gauge configured in a similar manner to the firing pin inside a real PMN-1 mine so as to most accurately determine if the mine would have triggered in a real-world situation. The testing system will also need to have on-board electronics to collect and transmit the force data to a computer located a safe distance away from the drop zone.

Item	**Cost**	**Notes**
UAS	\$4,000	All-in estimated cost for the drone and all components.
Vision	\$100 - \$500	Vision board and camera with lenses.
Testing Mine	\$70 - \$100	Construction materials and electronics.
Extra Parts	\$200	Extra propellers and electronics for repairs.
Totals	\$4,400 - \$5,000	

Table 2.1: Component cost breakdown.

2.4.2 Availability of Components

The idea of the project is that it is not tied to any specific components. All that is required to construct a working system is a sufficiently powerful UAS with batteries and chargers, a payload carrier, a vision system, a laptop or tablet computer, and payloads. The payload carrier that is developed as part of this project will be designed to be easily manufacturable without professional-grade machine tools. The payloads themselves are targeted to be constructed from modified drink bottles or similar containers, meaning that they will be available in copious amounts in most parts of the world. The wide availability of these components means that the system should be very widely available and usable in the areas that need it most.

2.4.3 Advantages

There are several advantages to this design over the current commercially-available solutions. The first is cost and availability. The proposed design does not require any licenses in most countries nor special training in order to operate. It simply requires the materials to build the UAS, payloads, and mine markers. It is affordable enough that a small municipality could construct a system and deploy it in their town to remove mines cheaply and efficiently.

The second major advantage is repairability. Replacement parts for most commercially available drones can be ordered directly online or the drone can be sent back to the manufacturer for repair in a timely manner. Self-done repairs do not require any large amount of technical skill, meaning that the groups using the system should be able to maintain the system reasonably easily. In addition, this proposed solution is cheaper than the current solutions by multiple thousands of dollars; if it is completely destroyed the system is cheaper than any commercially available system to replace.

The system is also customizable to suit the specific needs of the group who is using it. The platform designed for this project is fully extensible. Changes can be made to the existing design to suit a diverse set of needs. It could, for instance, be modified to carry different numbers and types of payloads or different cameras and sensors. This however could require skill sets that are less likely to be readily available in areas where the drones are most likely to be deployed.

2.4.4 Disadvantages

The major disadvantage to the system is durability. UAS's are designed to be light to maximize payload capacity and flight performance, meaning that a rogue piece of shrapnel can more readily damage the drone than one of the armored commercially-available systems.

The system also does not have a method of identifying land mines; each mine must be identified and marked beforehand. Ideally, a second robot traversing the minefield would mark all mines beforehand, but the total demining performance of this system is dependent on the other robot.

The other main disadvantage is that this system is designed to target only the PMN-1 and similar mines. Other types, such as blast-resistant mines, may not trigger and detonate due to an from a small airborne payload. The system can perhaps be modified to target these alternative mines, but the design outlined in this project does not take into account targeting these other types of land mines.

2.4.5 Constraints

There are three types of constraints for this project: legal constraints, construction and cost constraints, and testing constraints.

The first set of constraints deal with legal issues. Federal and state laws within the US mandate that a pilot must always have a direct line of sight to the UAS and must always be looking at it. As a results, testing this system around buildings will be difficult to do without breaking laws, so all testing should be done in a large, open field. The pilot must also be able to instantly take control of the UAS and disable the autopilot. This is a safety feature that must be implemented in the event that hardware or software fails, the area of flight is no longer safe, the UAS risks injuring a person, or the UAS begins to act dangerously and unpredictably. The UAS must also be kept under 10kg, otherwise it requires a license to fly. Federal laws also prohibit any projectiles being fired from a UAS, which means that the projectiles used in this project must be dropped and cannot be propelled by any forces other than gravity.

There are two main constraints for the construction of the UAS: materials and cost. First, the UAS will most likely be built using carbon fiber for the frame. The strength of carbon fiber will be strong enough to provide a steady frame, but its light weight will limit the weight ratio of frame to payload, thus allowing the UAS to carry more. Unfortunately, there are limited tools or equipment available on campus to work with carbon fiber. As a result, the frame of the UAS must be bought from an online retailer. While this option is more expensive, it does fit into the goal of a widely available project. If the end result of this project required access to carbon fiber machining tools, this demining solution would not be available to many towns. The other main constraint is its cost. Building a multi-rotor UAS with a carbon fiber frame is not cheap and is estimated to cost around $4,000 to complete.

There are three constraints for testing this project. First, a ground station is required to monitor the UAS at all times to communicate and report information such as position, velocity, or system state. A commercial system that supports telemetry data such as the APM, Pixhawk, or DJI series of flight boards can easily accomplish this. This ground station should be cross platform and work on as many devices as possible. A large, open field to test in is also required. There should be few obstacles and nothing that could be damaged by a UAS or the payload falling on it. Ideally, a test landmine should be able to be deployed in this field as well. The last constraint is critical. This project is a UAS demining project, not a mine detection project, so the landmines should already be marked. The method and precision of marking have not yet been settled upon, and the required precision will be determined via testing with the constructed testing mine.

2.4.6 Challenges

The project presents a number of interesting challenges that will have to be dealt with during its development. The first of these challenges deals with identifying and tracking the target. Simple GPS markers will not be able to pin-point the location of the target, since GPS is only guaranteed to be horizontally accurate to 3.5m (Force, 2014). This level of accuracy is not enough to be able to accurately home in on a mine and drop a payload with any degree of accuracy. Therefore a flag system will need to be designed to mark the mines visually, and have the UAS home in on it using a camera system. This brings about its own set of challenges, as homing in on a visual target from the air with a drone platform is no simple task. That said, research has been done that shows that it is possible to detect mine-like objects from the air using a UAS, which means that it should be possible to accomplish this task(Rodriguez, Castiblanco, Mondragon, & Colorado, 2014).

Another challenge is carrying and controlling multiple heavy payloads and ensuring that the drone stays airborne and stable after deployment. Each time the UAS releases a payload its distribution and amount of mass changes, meaning that control software will have to be developed that can accommodate for this change in mass and quickly correct to keep the drone in the air and continue normal operation.

The payloads also need to be dropped very accurately. This means that their aerodynamic properties need to be studied and optimized, so that a determination can be made as to what the best payload design is. Along with this the possible flight paths need to be studied so as to understand which ones yield the best targeting accuracy and the least danger of the drone being damaged. The UAS can either deploy the payload on a ballistic trajectory while the platform is in motion in order to move horizontally out of the mine's blast cone, or it can drop the payload straight down and move directly upwards to escape the blast cone. These and other flight paths need to be studied in order to understand which yields the best performance.

Chapter 3

Methodology & Experimentation

3.1 Testing Device

Before any construction of the UAS system could begin, a series of tests were conducted to prove that a payload dropped from a reasonable height could detonate a PMN-1 landmine. To conduct these tests in the most accurate manner, a simulation landmine was created that matched the detonation specification of an original PMN-1 landmine. The triggering mechanism was replaced with a strain-gauge to measure impact forces.

3.1.1 Design and Construction

The test landmine was constructed using a 5 inch PVC tube for a body in place of the metal canister that the PMN-1 would have. The top of the test landmine was made out of a sheet of rubber, mimicking the sheet of rubber on top of an actual PMN-1 landmine. This sheet of rubber was stapled in place around the top of the practice landmine while the sheet was held taut but not strained or stretched while it was fastened. A small round sheet of plywood was placed underneath the rubber to simulate the PMN-1 pressure plate. A strain gauge, used to measure impact forces, was placed against the underside of this sheet of plywood and held in place below it by a second sheet of plywood. A small hole was cut in the second sheet of plywood to allow the strain gauge to flex accordingly. This second sheet of plywood was attached to the main body of the test landmine with 4 screws. The plates and strain gauges can be see in Figure 3.1. Between this second sheet of plywood and the bottom of the simulation mine lay a second control strain gauge, Arduino uno with Xbee, RC filter circuit, and a 9V battery. The practice mine internals and outside can be seen in Figure 3.2.

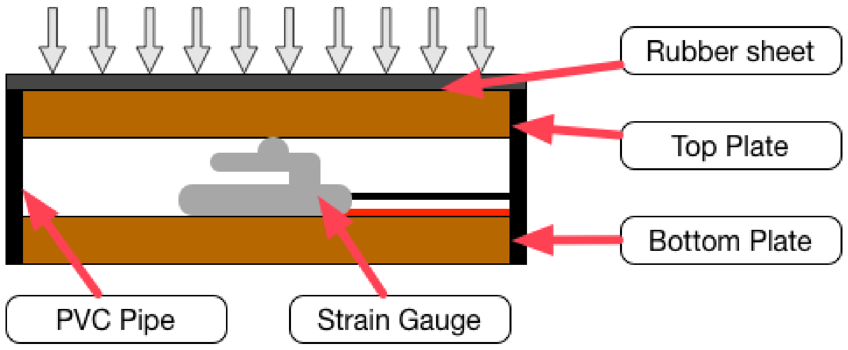
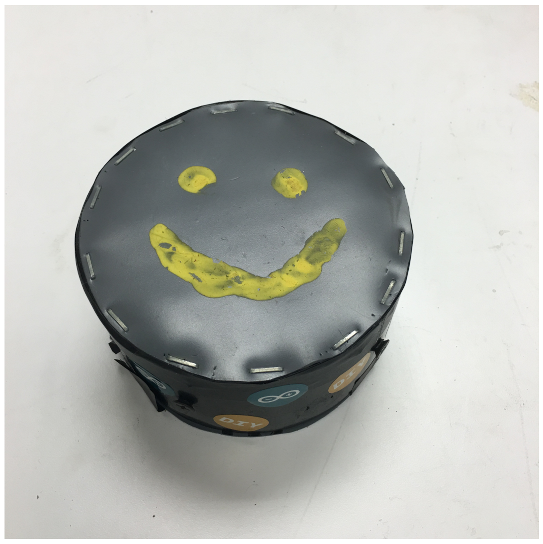
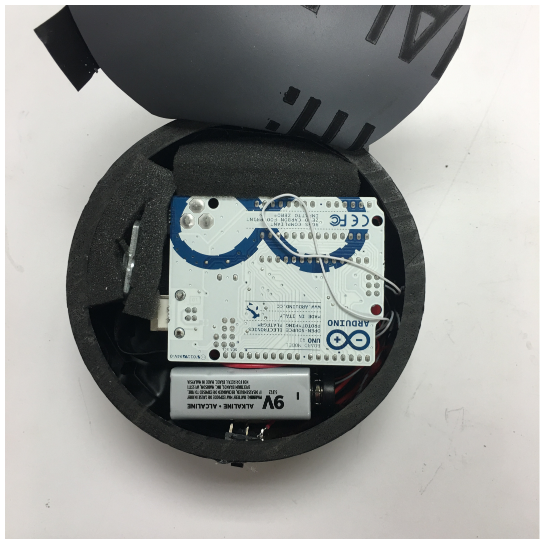


Figure 3.1: Internal configuration of strain gauge and plates.



(a) Top view of the practice mine, showing the impact plate.



(b) Mine with bottom cover removed, showing some of the electronics.

Figure 3.2: The practice mine.

To measure impact forces on the pressure plate of the simulation mine, two strain gauges and a differential amplifier were used. Two strain gauges were used in order to remove environmental effects from the impact force calculations. One strain gauge received the impact force from the projectile and the other would serve as a control, so that the data would not be affected by temperature, humidity or other environmental factors. This second control strain gauge was padded in foam and placed at a 90 degree angle offset from the other strain gauge so that the force from the projectile could not affect the control strain gauge. Both signals were first passed through an RC circuit with a cutoff frequency of 33.86Hz to remove any high-frequency noise and then passed to a differential amplifier. This differential amplifier has a gain of 2,246 corresponding to an output voltage on the magnitude of 0-5V for a force between 0-100 newtons. The amplified analog signal was then passed to an Arduino for conversion into Newtons and recording. This circuit can be seen in Figure 3.3. An xbee wireless radio was connected to the Arduino Uno and used to transmit the converted data to a receiver connected to a laptop. The entire system was powered by a single 9V battery.

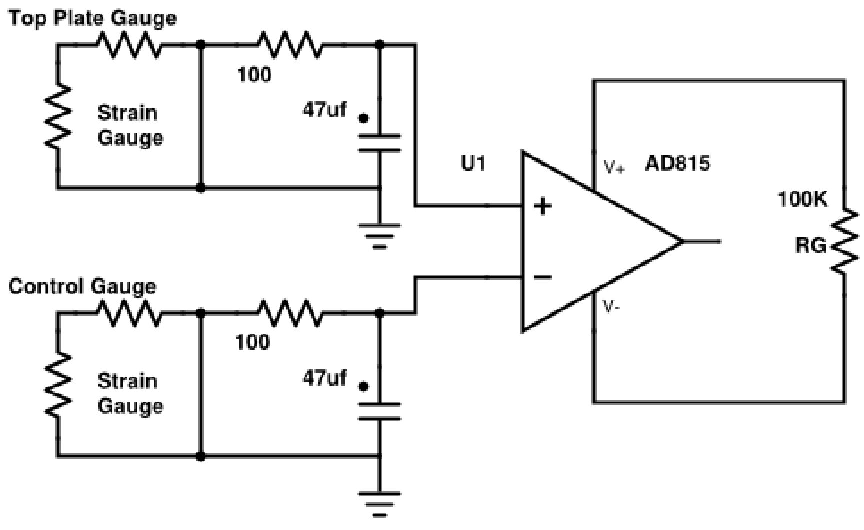


Figure 3.3: Strain gauge and filter circuit diagram.

3.1.2 Calibration and Code Issues

Before any testing could begin, the simulation mine needed to be calibrated. To calibrate the conversions function, 25 masses ranging from 0-10 kg were placed on the pressure plate of the simulation mine. The corresponding analog values for each of these weights were recorded in a csv file and the relationship was plotted. The linear relationship was noted and a line of best fit was calculated and plotted. The relationship and line are shown in figure 3.4. With the relationship between input weight and output analog value, slope, and y-intercept known, a function to calculate the impact force was determined. The function is as follows:

$$Weight = (sensorreading - offset) * 15.376$$

In the beginning, it was assumed that the offset (y-intercept) was constant, but it was quickly noted that the offset changed between tests, affecting the calculated weights. To account for this, a calibration routine was implemented in the arduino sketch that determined the no-weight offset before testing began. Despite the offset shifting, the slope of the linear relationship (how much a change in input affected the change the output) did not change, requiring no further calibration.

One issue encountered while testing the mines was the loss of data due to transmission speed. While developing the impact force logging code, the arduino would analog-read

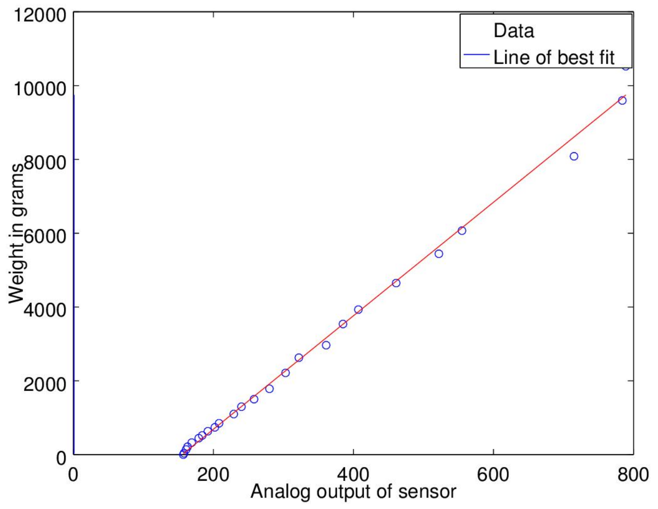


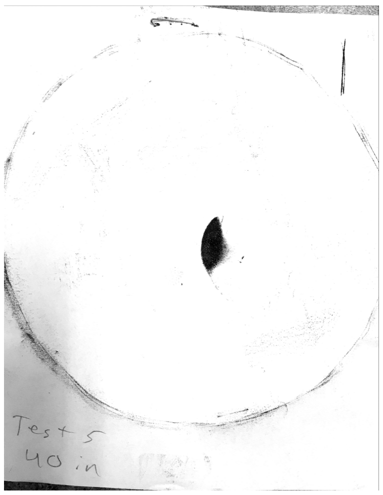
Figure 3.4: Calibration curve for testing device

the strain gauge and then immediately transmit the data over the xbee radio connection. Unfortunately, the serialization and transmission of the data took a significant amount of time. An entire impact only took approximately 0.05 seconds. The combination of these two facts meant that a significant (greater than 80%) portion of the data was not recorded. To work around this, the data was buffered in the arduino's ram and then transmitted all at once after the impact.

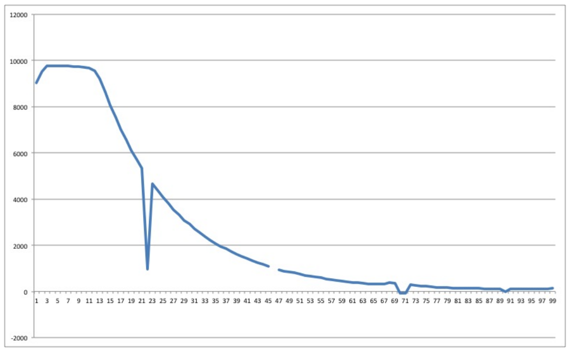
3.2 Testing

The goal of testing was to determine the impact force generated by each of the projectile considerations. Projectile options were a full water bottle and small bag of sand. A projectile could 'detonate' the simulation mine if the impact force exceeded 50 N. This impact force is the rated trigger force of the PMN-1 landmine as determined by the research.

To begin, each projectile was dropped from a height of two feet and increased until the recorded impact forces exceeded 50N. Once the necessary height had been determined, carbon paper was placed upon the top of the mine to record the impact location and the impact data was logged and plotted using Octave. An example of of the carbon paper and impact plot is shown in figure 3.5. Through these tests, it was empirically determined that the projectiles must be dropped from a distance of approximately 40 inches or more.



(a) Carbon paper showing a large mark at point of payload impact



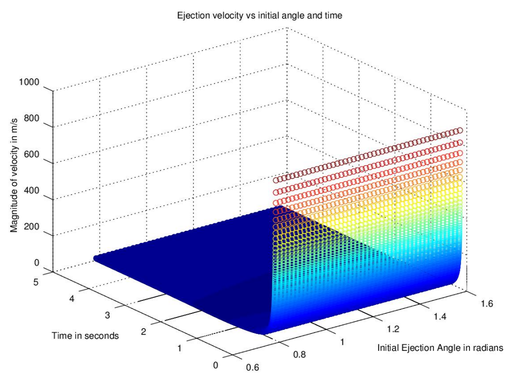
(b) Drop test force from payload impact

Figure 3.5: Payload drop data for drop on testing mine (dropped from 40in)

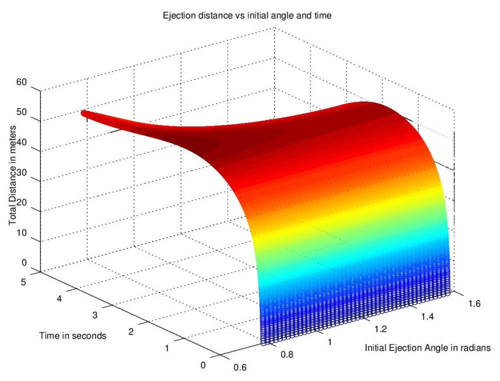
3.3 Shrapnel Calculations

To be an effective demining solution, the UAS system needed to be reusable and durable. Shrapnel from landmine detonations could damage a UAS, requiring repairs that are both costly and time-consuming. The necessary mobility of a UAS to avoid shrapnel needed to be determined before a craft could be selected. Using information reported in the Australian Department of Defense Paper, *Evaluation of a Silent Killer, the PMN Anti-Personnel Blast Mine* (Swinton & Bergeron, 2004), a model was created to predict the distance and speed of the particles as a function of both time and inclination.

The shrapnel was modeled as a cube of 5mm in side length with a mass of 0.2 grams. A coefficient of drag of 1.05 was used and the ejection velocity was initialized to 900 m/s as measured in the paper. The Octave code below was written to model the shrapnel with the given inputs in discrete time using time intervals of 0.001 seconds. The position and velocity of the shrapnel were calculated at each time interval and plotted. Additionally, these calculations were repeated at every whole degree of inclination from 0 - 45 degrees to accurately model the shrapnel in all possible directions. According to the previously referenced paper, after a 45 degree inclination, there is nearly zero shrapnel, meaning the model only needed to model 0-45 degrees. Below are the figures displaying the results of the model.

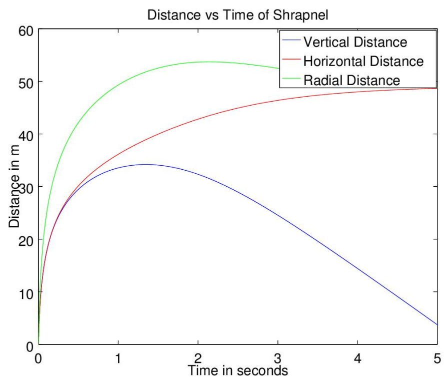


(a) Ejection velocity vs. initial angle and time

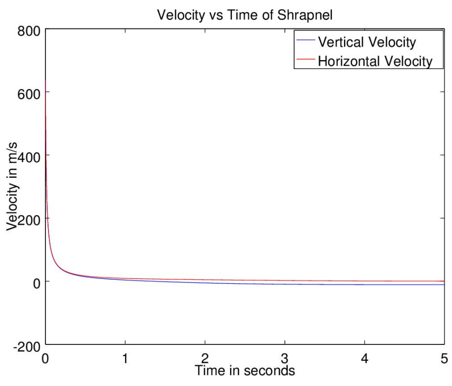


(b) Ejection distance vs. initial angle and time

Figure 3.6: Shrapnel calculation graphs for mean steel band shrapnel shards.



(a) Radial Shrapnel Distance vs Time



(b) Shrapnel Velocity vs Time

Figure 3.7: Shrapnel calculation plots for distance and velocity vs time.

Assuming the payload would be dropped from the UAS, while hovering, at approximately 20 feet of altitude, this gives the UAS 1.12 seconds to accelerate either far enough away to experience minimal shrapnel damage or, better yet, out of the shrapnel cone entirely. After about 0.2 seconds after detonation, the shrapnel velocity decreases from about 900 m/s to 10 m/s, a speed that would result in less UAV damages. At this time, however, the shrapnel already covers a distance of 30 meters radially (approximately equal for all ejection angles). Thus, the UAV must accelerate 30 meters away from the detonation point (while still within the shrapnel cone) within 1.32 seconds to reach this low-velocity shrapnel impact zone.

To reach this point by accelerating only vertically, the UAS would need to accelerate at $27.55 m/s^2$ (see equation 3.1), an unachievable acceleration. The UAV is unable to 'outrun' the shrapnel vertically, however it can escape the blast cone (45 degrees). Assuming the UAS drops the payload at 20 feet (6 meters) from a hover, it would need to travel the same distance in approximately 1.12 seconds to clear the 45 degree shrapnel cone. This corresponds to an acceleration of $9.72m/s^2$, a much more reasonable requirement that has been experimentally verified.

$$a = \sqrt{\frac{2\delta X}{t^2}} = \sqrt{\frac{2(30m - 6m)}{(1.32s)^2}} = 27.55m/s^2 \tag{3.1}$$

3.4 UAS Selection

3.4.1 Airframe

The UAS platform for this project was shared with another team and will be shared with future teams. As a result, a versatile platform that would meet all team requirements was needed, so a list of constraints was compiled to select an appropriate UAS. The final list of constraints, in no particular order, is below.

- UAS must be able to carry 4 to 6 kg
- UAS must be able to fly for 10 minutes under load
- UAS must accelerate horizontally at a minimum of $10m/s^2$
- UAS must have mount points for custom payloads
- UAS must be able to fly both autonomously and under human control
- UAS must be able to communicate with a ground station
- UAS must be easy to repair when necessary
- UAS must be upgradeable and support additional hardware/software for future projects
- UAS must not cost more than the department was willing to pay for

A list of potential airframes and flight controllers that matched the constraints above was compiled, and only two commonly available airframes and one set of flight controller hardware matched the constraints.

The first frame was the DYS v800 Heavy Lift platform. This met all the constraints but had a 1-2 month shipping time and spare parts were difficult to find. There was also no easy way to contact the company should something break or go wrong.

The second option for the airframe was the DJI Spreading Wings S1000, which met all the constraints and had a very easy way to contact the company should the team need to. In addition, spare parts could be ordered from a third-party vendor. This airframe was chosen for the project.

3.4.2 Flight Controller

The final component to select in order to construct the drone was the flight controller. This controller needed to be able to support the eight-prop airframe that had been chosen, and allow us to have full autonomous control over the UAV.

The first option considered was the DJI family of flight controllers (DJI, 2016). Since a DJI airframe was chosen, this was the ideal place to start. DJI makes 3 flight controller families: the Naza-M V2, the WooKong-M, and the A2. All of these offer the same base level features, with prices going up from $299 for the Naza-M V2 to $1,299 for the A2. The A2 offers the largest set of features and the most expandability. For sensors, it can integrate with an included IMU as well as a compass module.

The other option in consideration was the 3DR Pixhawk (3DR, 2016a). The Pixhawk has the same basic features as the A2, along with a few benefits. It has 2 input telemetry ports for control (one for the pilot controller, and one to connect to a secondary controller); expandable SPI, CAN, and I2C ports; as well as an integrated ADC. It can interface with a combined compass and GPS module as well, which allows for full autonomous flight. The Pixhawk also has a fully open-source development kit called Dronekit which allows for full drone control via a telemetry port from an external controller (3DR, 2016b). The Pixhawk kit cost in the neighborhood of $400.

The final choice was the Pixhawk. It provided the level of autonomy that was required at a price point that was within the scope of the budget outlined for this project. For a secondary controller, a Raspberry Pi 2 Model B was chosen. It had the serial connection required to run the Dronekit libraries as well as a Wi-Fi link to allow for monitoring of the autonomous routine from the ground.

3.5 UAS Setup

Due to the fact that the flight controller and electronics system is not out of the box compatible with the DJI S1000 airframe, several modifications had to made to the airframe and electronics to allow the systems to work together.

3.5.1 Frame

The first change that had to be made for the airframe was to change the direction of every motor. As the Pixhawk has these values hard coded in, and the direction of rotation of a motor is usually very easy to change, these values are hard coded into the Pixhawk. Usually this is accomplished by swapping two wire connections on the output of the ESC. This is not possible on the S1000 due to the construction of the ESC, so each arm had to be swapped with another arm that had a motor spinning in the opposite direction. Two arms on the S1000 were marked with red plastic and red LEDs to show which direction is forwards. Due to the length of the wires and the fact that the Pixhawk should be centered in the airframe, the Pixhawk had to be rotated by 180 degrees. This means that the two red colored arms also had to be moved to match the new forwards direction.

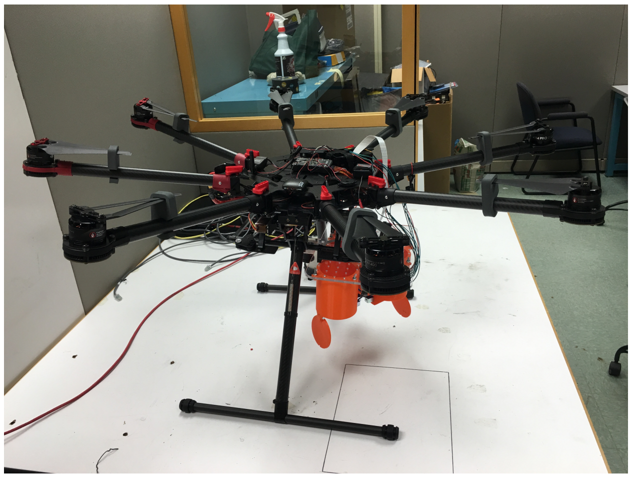


Figure 3.8: Side view of airframe, wings extended.

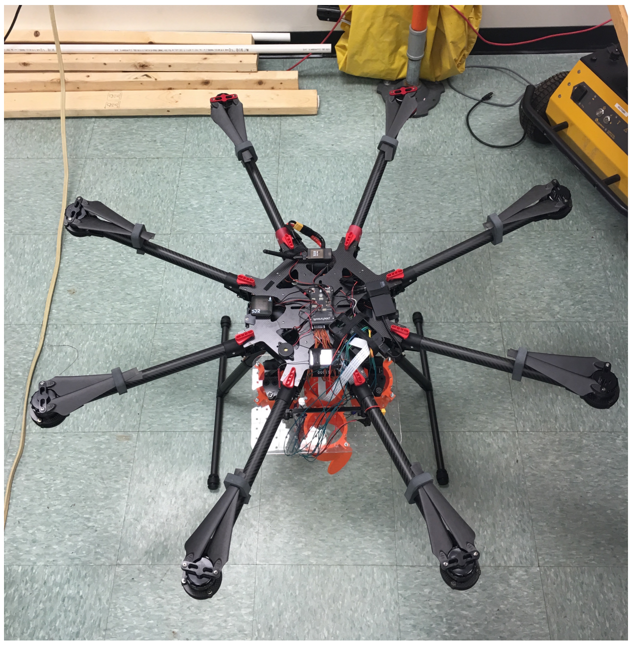


Figure 3.9: Top view of airframe, wings extended.

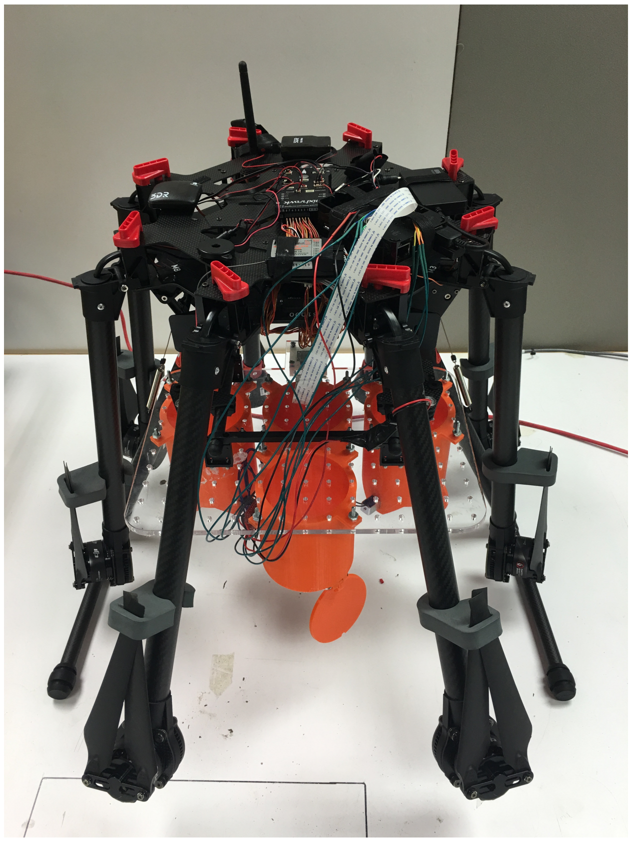


Figure 3.10: Airframe with wings folded down.

3.5.2 Wiring

Normally the wires from the ESCs would go directly to the same numbered port on the flight controller. Due to 3DR and DJI not using the same standards, this is not the case. See below for details.

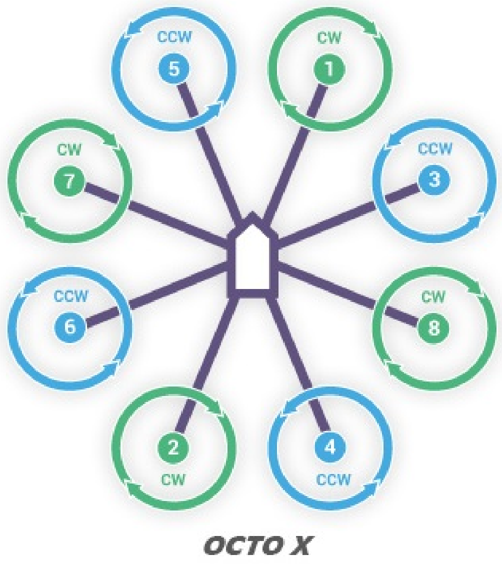


Figure 3.11: Directions and connection information for an OctoCopter using a Pixhawk, from http://copter.ardupilot.com/wiki/connect-escs-and-motors/

3.5.3 Controller and Radios

The Pixhawk has two methods of user control, one from a hand held radio controller and one from a telemetry radio. The telemetry radio connects to a second identical radio which is plugged into a device to control the UAS. Programs to interface with the telemetry radios are available on Android, Windows, Linux and OSX. On Android, the app is called Tower. On all other systems it is called Mission Planner. These systems need very little calibration or setup. The other option, a hand held radio, requires much more setup but allows for a much greater and finer system of control. Almost all of the setup is programming the hand held controller. For a FLYSKY FS-TH9X controller, the stick set is Mode 2, the elevator channel is reversed, channel 5 is set to the throttle hold switch, and channel 8 to the gear switch. An additional programmed mix must be added and set as follows: MIX1 - ACT, GYR, FLP, 000, +100, -100, ID2. MIX2 - ACT GYR FLP 000 000 000 ID1. MIX3 - ACT GYR FLP 000 -100 +100 NOR. The first three refer to the source switch of the base signal, and the fourth specifies that the mix should take no input from this switch. The fifth and sixth modify how much the additional switch modifies the signal, and the seventh is this additional switch. ID1 and ID2 refer to positions of the 3 position switch and NOR is for when there is no input from it. This creates a system where channel 6 will output a high, medium or low value based on the position of the 3 position switch. To connect the receiver to the Pixhawk, a PPM Encoder is needed. This converts the several wire PWM signals to a single wire PPM signal, which is what the Pixhawk uses. The numbered outputs from the radio receiver match with the numbers on the PPM Encoder wires, with the exception that channels 5 and 6 must be swapped to ensure the wires are routed correctly.

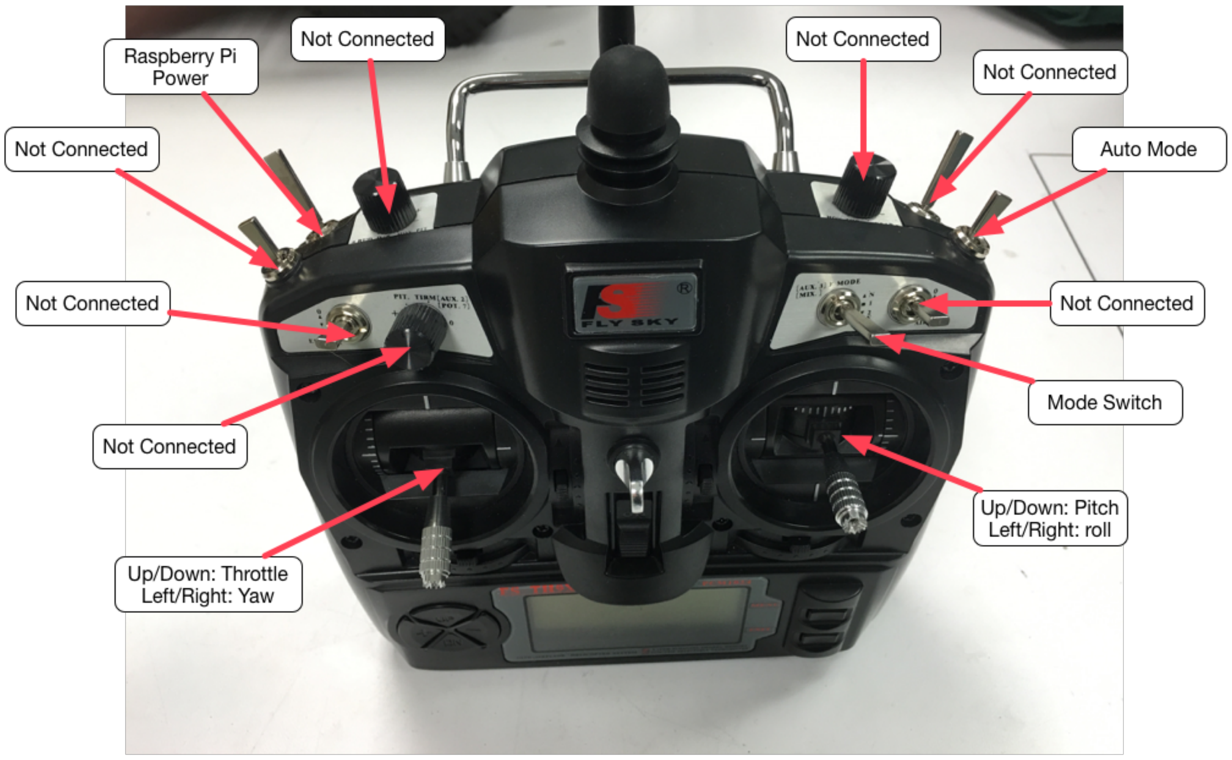


Figure 3.12: Annotated view of controller, showing function each control is mapped to.

3.5.4 Ground Control Software

Mission Planner is the best Ground Control Station Software that the team was able to find. The current configuration has mapped three flight modes to the Pixhawk; the first is manual control, the second is Loiter and the third is Return to Launch. The only Failsafe that is configured is for when the handheld transmitter is turned off or disconnected as the other failsafes require hardware that is not present or has been replaced for various reasons. There is also a geo-fence that is enabled, so if the UAS flies too far from the test area it will automatically return to it. Channel 5 is set to flight mode, channel 6 and 7 to nothing and channel 8 to auto routine, which will run a user generated autonomous mission if one is setup.

3.5.5 Failsafes

The only failsafe in place is for a controller disconnect. In the case of this event, the UAS is set to Return to Launch. There are failsafes available for battery voltage and current, but as the battery in use is higher voltage and capacity than supported, the software can not enable this failsafe. Instead, a battery alarm has been attached that begins beeping if the battery drops below a certain voltage. The voltage that triggers the alarm is programmable and set to 3.6 volts per cell. FAA rules require that a pilot must be able to take manual control away from a UAS at any time. In order to do that, channel 6 has been routed to a relay. This relay can turn off the power to the Raspberry Pi at any time, and after the pilot has powered off the Raspberry Pi and changes the flight mode switch, the UAS is under pilot control.

3.5.6 Power and Battery

The Pixhawk runs off of 5 volts, and the batteries in use output 25-18 volts. In order to power the Pixawk off of the batteries, a power module was installed in between the battery and power distribution board that has a buck converter on it which outputs 5 volts. It is an off the shelf part that can be bought online. This power module also has analog voltage outputs for the voltage remaining in the battery and the current draw in real time, but as the Pixhawk only supports batteries of up to 17 volts, neither of these are available for use. The power module in use is not the one that came with the Pixhawk, as that power module also only supports up to 17 volts. In order to power the Raspberry Pi and additional electronics such as the payload bay, two additional buck converters were needed. These are the same units used to the power the Pixhawk. One was needed to power the Raspberry Pi itself, and one for the powered USB hub that was used for high powered wireless adapter. These buck converters must be installed in parallel with no current draw on the high voltage passthrough, as installing them in series results in the buck converters being destroyed. This is most likely due to the fact that the buck converters use current sense resistors, and adding two of these in series will give inaccurate readings.

3.6 Payload Development and Testing

When designing possible payloads, four objectives were kept in mind: the payload should should weigh about 0.33 kg (empirically determined from simulation mine testing), the payload should be small enough so that the UAS platform could easily carry at least three (project constraint), the payload should be cheap and easy to manufacture (project constraint), and the payload should be able to be dropped accurately (project constraint).

The first payload design was a water bottle with fins glued onto it, shown in Figure 3.13. This design was originally very attractive, as it was a cheap, readily-available solution that was both aerodynamic and matched the desired weight requirement. However, this payload design proved difficult to design a mount for and was rather large. In addition, during testing, a significant amount of tumbling was observed, indicating it would not be as accurate of a solution as desired.

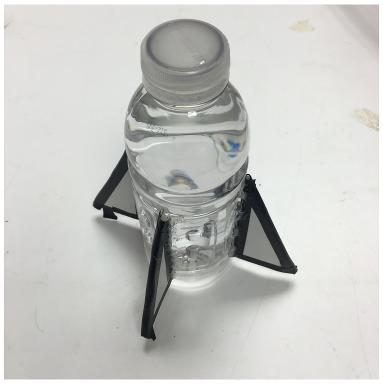


Figure 3.13: Water bottle based payload design.

The second design aimed to improve on the tumbling issue; it was roughly half a kilogram of metal nuts formed into a sphere inside three layers of cling wrap. This proved to be very effective as it would deform when it hit the target and distribute the area over which the force was distributed. This was a problem with the first projectile design as the projectile would tumble and as it was a bottle cap hitting a plate at an angle, the area over which the force was distributed was very small. Due to the fact that the strain gauge, and trigger of the real PMN-1, is in the center of the rubber plate the first projectile would have to hit dead center to reach the required force to detonate the mine. With the second design the projectile would spread the force over a wider area and in testing needed less accuracy to set off the practice mine in small scale tests. The final design, shown in Figure 3.14 simply replaced the metal nuts with sand so as to not add more shrapnel to the explosion and to be more eco-friendly when the payload bag ruptured in outdoor testing. Water may also be a viable material to make the payload out of, but the team did not test this as the practice mine was not waterproof.

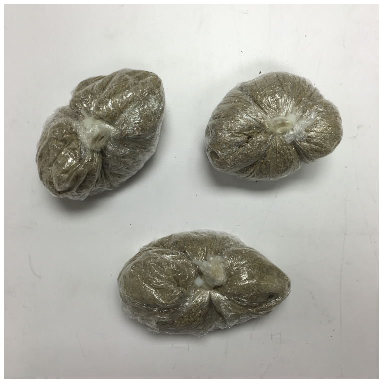


Figure 3.14: Final payload design, based on plastic wrap filled with sand.

3.7 Payload Bay Design

The first payload bay was designed around the idea that the UAS should be able to automatically reload, or do so with the help of another robot. The design was based on a revolving door and consisted of a large hollow cylinder made of laser-cut acrylic. The top and bottom plates were held together with spacer-nuts and the bottom plate had a cutout for the payload to drop from 3.15. A motor with an encoder on it was used to power the revolving door and force payloads out of this hole. A total of 5 payloads could be carried. Another mechanism was to be designed for reloading, but as this payload bay design was scrapped after the first prototype this device was never designed. The reloading mechanism was going to be a slide that mated with the opening in the payload bay, down which payloads would slide. The payload mechanism would run in reverse and this would load the payloads into place.

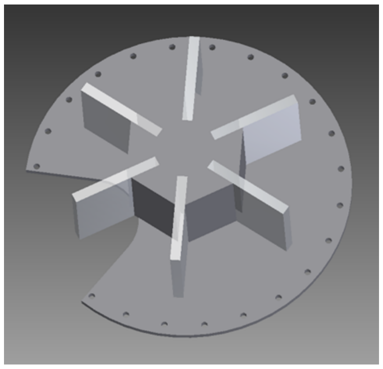
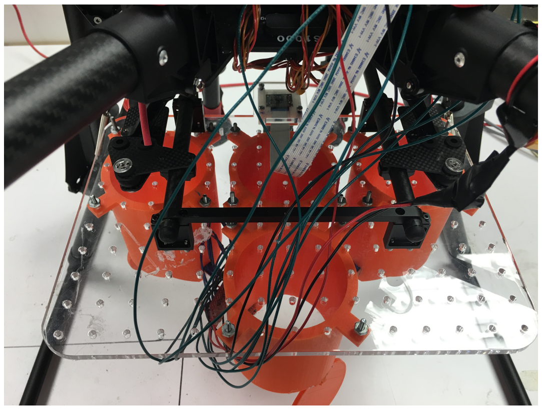


Figure 3.15: Original payload bay design.

This design was ultimately scrapped due to several reasons. The first was that the torque the motor would put out was enough to move the payload, but if the payload got caught on something it could tear or the motor might stall. The next reason is that while the payload would safely exit the payload bay and fall, the exact time that it would do this was not certain and could be roughly estimated at best. This was due to the fact that different shaped payloads could sit differently within the chamber or be pushed out of place at a different time based on the angle of the UAS at that time. This payload bay design was also very hard to mount to the UAS and would have interfered with the landing gear.

The final payload bay was designed to carry four projectiles, each of which could be released separately. Each payload was housed in a 3D printed tube with a hinged door and latch to keep the payload secured. The latch was a solenoid that was controlled through a standard 1 amp h-bridge driver connected to the GPIO pins on the Raspberry Pi. Each of these tube assemblies was screwed to a laser cut acrylic plate, shown in Figure 3.16. This plate also had mounting holes for the Raspberry Pi camera that was used for aiming. The system was designed so that there was a known offset from the camera to the payload tubes that the Raspberry Pi could account for when it would aim for the target. This plate was then attached to the camera gimbal mount points on the UAS, and the gimbal position and battery tray were moved to keep the center of mass roughly centered in the UAS. The final payload bay and the CAD model that accompanies it can be seen in Figure 3.17.


(a) Top view of mounting plate.

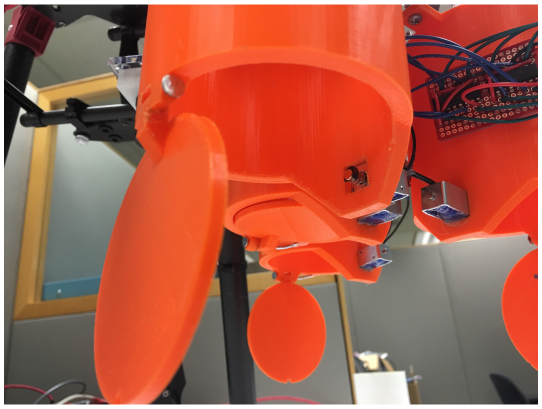
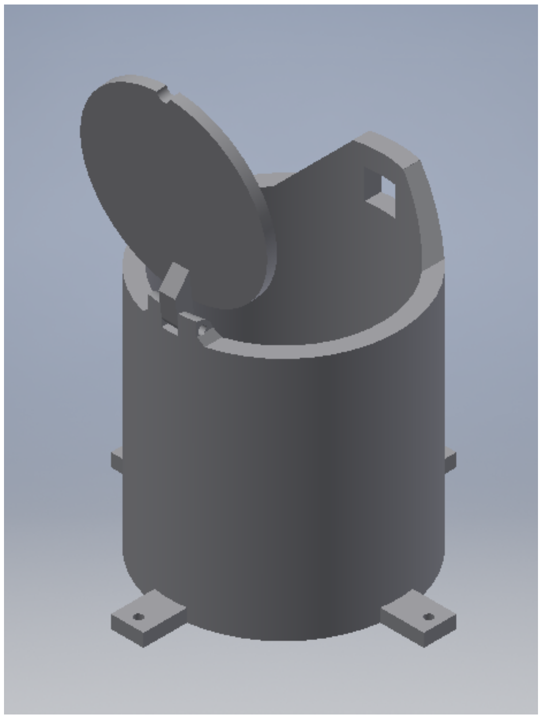

(b) Payload tube door, showing the solenoid assembly and the trapdoor itself.

Figure 3.16: Payload bay trapdoor and mounting plate.


(a) Final payload bay design (open, CAD).

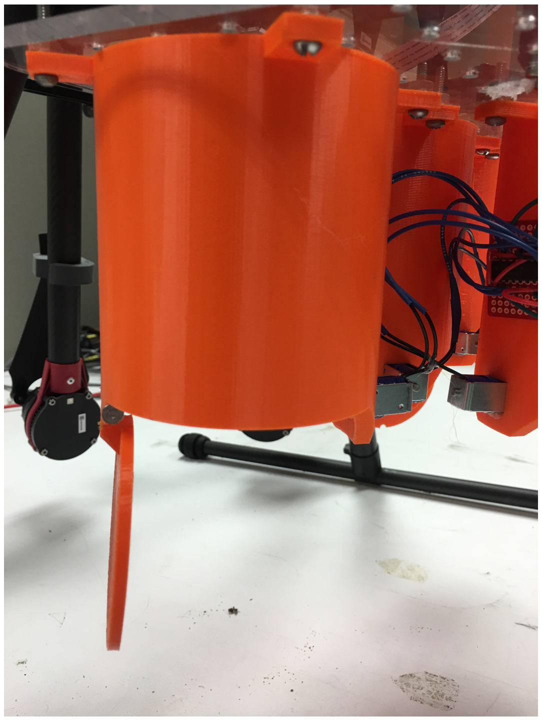

(b) Printed payload bay mounted on UAV.

Figure 3.17: Final payload bay design, showing both the CAD model and the finished product mounted on the UAV.

3.8 Vision System

In order to identify the marked land mines on the ground, an adequate vision system needed to be developed. Having previously chosen to use a Raspberry Pi to interface with the drone controller on the UAS, an ideal candidate for the camera was the Raspberry Pi Camera. This camera is a 5MP sensor capable of 1080p video at 30 frames per second (FPS), or 720P at 60FPS. The camera interfaces with the Raspberry Pi through a special ribbon cable, making interfacing with it faster than an interface with a USB webcam through the Video for Linux 2 (V4L2) library. The camera module itself is very inexpensive and provides reasonable performance.

3.8.1 Lens Selection

The built-in camera lens provided reasonable performance up to a distance of 15 feet using 480p resolution. Some level of magnification would be required to be able to spot targets consistently at our target elevation of 20 feet. To achieve this magnification, the original RasPi camera lens was replaced with a new lens to simplify designs.

The new lens needed to meet three criteria. First, the lens must be relatively cheap. This project required many components and the budget is limited, thus the budget for the lens was capped at $50. Second, the lens must have a reasonable field-of-view (FOV). It is easy to find lenses with the proper magnification, but they constrain the FOV to +/- 20 degrees, strictly limiting the area that could be scanned with the camera. Third, the lens must apply the proper magnification to identify targets at 20 ft. To calculate the new lens' focal length, the following equation was used:

$$M = \frac{f}{f - d_0} \tag{3.2}$$

Where f is the focal length and d_0 is the distance to the object. The exact magnification required was unknown, but the required magnification with respect to the default lens was known:

$$\frac{M_1}{M_0} = \frac{20feet}{15feet} \tag{3.3}$$

Taking the ratio of the two magnification equations yields:

$$\frac{M_1}{M_0} = \frac{20feet}{15feet} = \frac{f_1 * (f_0 - d_0)}{f_0 * (f_1 - d_0)} \tag{3.4}$$

Assuming:

$$f_0, f_1 << d_0, \frac{f_0 - d_0}{f_1 - d_0} = 1 \tag{3.5}$$

Solving for f_1 yields:

$$f_1 = \frac{4}{3} * f_0 \tag{3.6}$$

The focal length of the default lens was 3.6 mm, thus the required new focal length was at least 4.8 mm.

Considering all three constraints, a $30 lens with a 6mm focal length and 48.1 degree FOV was chosen. The lens worked as expected and increased the viewing range out to 7 meters.

3.8.2 Vision Target Choice

The two main requirements for a target were that the target must be easily identifiable at 20 ft and approximately the same size as the landmine. Since the UAS cannot see the physical landmine (as it is covered by the target), it must aim to hit the center of the target. To assure that a solid target impact also is a solid landmine impact, the target should be about the same size of the landmine. This prevents issues with hitting the side of the target and not hitting the landmine.

Ideally, the target should be the same area as the landmine, but this unfortunately conflicts with the desire for the target to be identifiable at 20 ft. In preliminary tests, a target with a 6-inch cross-section (compared to 5 inches for the landmine) had spotty identification. When the cross-section of this target was increased to 8.5 inches, identification was much more consistent.

The shape of the target was arbitrarily chosen. The goal was to choose a shape that was easily identifiable using OpenCV. An OpenCV square identification algorithm was freely provided (see next section), so the shape of the original target was a square. The algorithm worked well, so the shape stayed a square.

A number of different colors were experimented with to try and increase the consistency of the target identification algorithm. The idea was that unnatural colors (yellow and red) would be easier to identify from background objects than natural colors (green and brown). While a human could easily distinguish a red square, the vision algorithm mostly relied upon contrasts to identify the outline of the square. After experimenting with many different color combinations, the target with the best identification rate was an 8.5 inch square target with a black electrical tape outline. This black/white contrast provided the necessary conditions for optimal identification. The final target design along with the timeline of development can be seen in Figure 3.18.

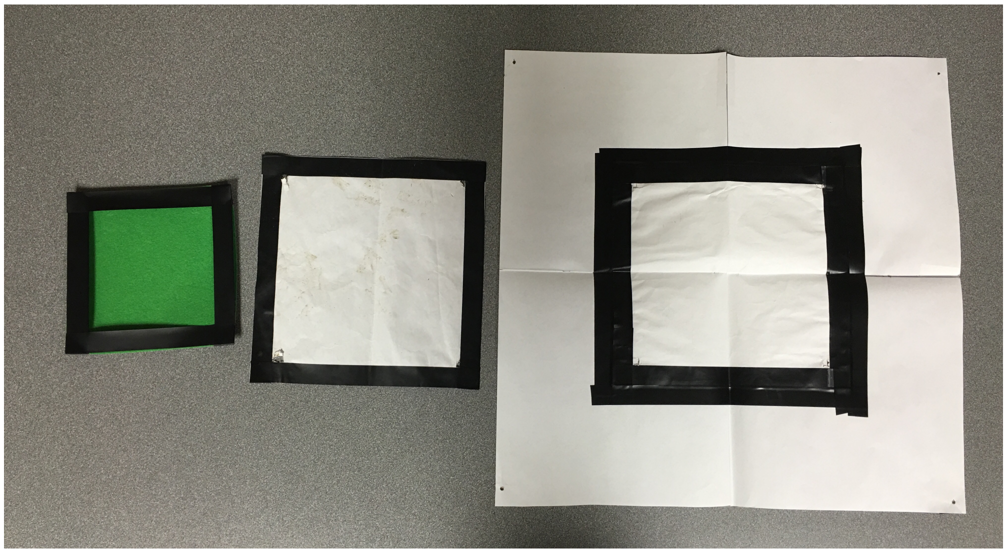


Figure 3.18: Target development progression, with the oldest on the left and the final on the right.

3.8.3 Algorithm Modifications

None of the team members had any experience with OpenCV, so research was done into identifying squares using OpenCV. Square identification code was freely provided by Adrian Rosebrock from his website *http://www.pyimagesearch.com* as an example. This code was downloaded and tailored to specifically identify the land mine targets.

Before modifications, the target identification algorithm would identify squares everywhere, from blades of grass to patterns on people's shirts. To fix this problem, a minimum size requirement for a square was required. Assuming that the UAS hovers 6 meters above a target, and that the camera viewing angle is 48.1 degrees, the camera views a horizontal area of

$$2 * 6m * tan(\frac{48.1}{2}) = 5.3m \tag{3.7}$$

The horizontal distance is represented by 640 pixels. An 8.5 (0.22 m) inch square target sits in this image, represented by square of specific pixel width and height. To calculate the pixel width of the square, the following ratio was used:

$$\frac{0.22m}{5.3m} = \frac{width}{640pixels} \tag{3.8}$$

Thus the approximate width of the target in an image in 28 pixels. The same principle is applied to the height of the target to achieve the same pixel values. To account for errors in altitude or viewing angle, a minimum size of 15 pixels for the side of a square was specified. This modification reduced the false positive target identifications to zero.

The second and more complex modification was implementing multi-core support of the algorithm to speed up processing times. This is discussed in much more detail in the Optimization Section (3.8.4).

With these two modifications complete, a final modification was needed: the image identification code needed to be integrated into the main control system. Instead of dissecting this algorithm and inserting it piece-by-piece into the flight control script, this algorithm was run standalone and passed important data to the control script through ports. A benefit of this script running standalone was that it was scheduled as a separate process from the main flight control code. Thus, the image identification algorithm could run at any speed (and block if necessary) and not affect the speed at which the main control code was running. This was important to maintain complete control over the UAS at all times.

While separate processes were extremely beneficial speed-wise, it also brought extra complexity in how shared data needed to be passed and managed. The image identification code needed to communicate whether or not a target had been identified and where. This data could not be directly accessed and the main flight control could not block and wait on data from the image identification code. As a result, an asynchronous data-passing method needed to be implemented. To accomplish this, a client-server port scheme was implemented. The target identification script would push data through a local port on the RasPi that would then be accessed by the flight control script. This is discussed in much more detail in the System Integration section (3.10).

3.8.4 Target Identification Algorithm

The target identification algorithm is rather straightforward. It is split up into three main steps.

First, a raw image passed as input to the algorithm is cast to grayscale and blurred slightly. The algorithm requires no color data, so casting the RGB value into a grayscale value reduces the amount to data that needs to be processed. Additionally, blurring the image slightly blurs out most of the 'weak-contrast' areas, benefiting the next step of the algorithm.

Once cast to grayscale and blurred, an image undergoes edge-detection and polygon generation. During edge-detection, sharp-contrast areas (the weak-contrast areas have been blurred away) of an image are identified and turned into lines. Then the lines are connected into approximate polygons by the polygon-detection function.

The last step is to determine if any of these polygons match a 'square' description. A square has three properties that are tested against. A square has an aspect ratio of 1, has four sides, and has a hull area ratio (bounding rectangle area/true area) of 1. To be considered a square, the polygon must approximately match these three criteria. Due to the imperfect viewing angle, lighting conditions, and other sources of error, it is very difficult to match all of these criteria perfectly. Thus, the aspect ratio of the polygon must be between 0.8 and 1.2, the number of sides must be between 4 and 6 inclusive, and the hull area ratio must be greater than 0.8.

If a polygon passes these three criteria, it is considered a square. A bounding rectangle is generated and the center of that bounding rectangle is calculated to represent the center of the square. This center information is then passed to the flight control script and the entire image is passed to the streaming script.

A visualization of the entire algorithm is shown below in figure 3.19 and three image-processing steps are shown below in figure 3.20

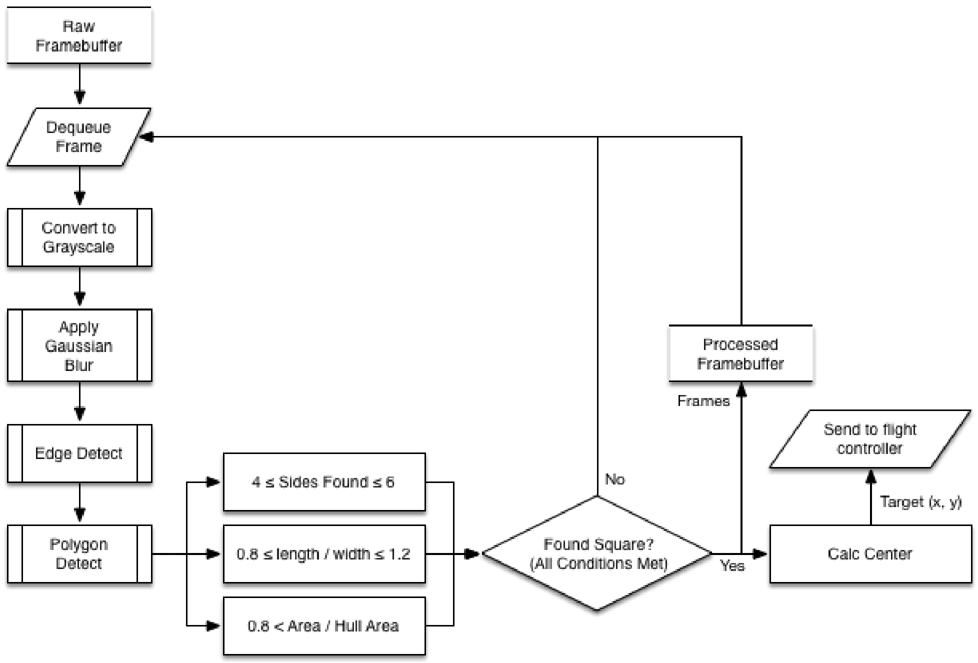
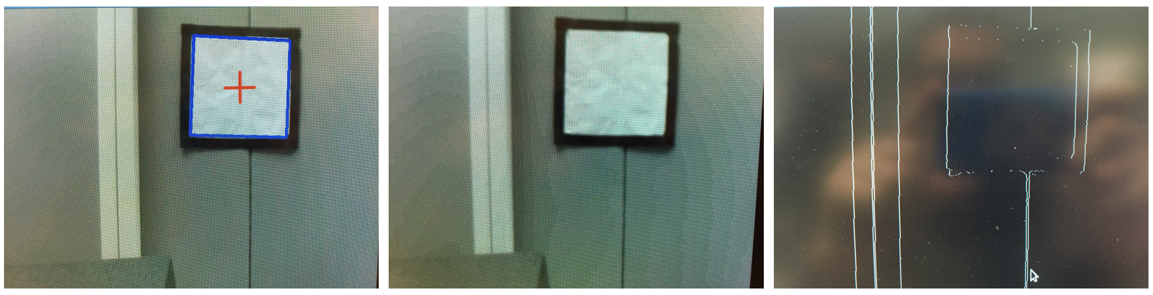


Figure 3.19: The target identification algorithm



(a) Identified (b) Grayscale and Blurred (c) Edge-detected

Figure 3.20: Image Processing Steps

3.8.5 Optimization

The most important modification to the target identification algorithm was the implementation of multi-process support. When first run on the Raspberry Pi, it was noted that the algorithm consumed one CPU core completely and output images at approximately 2-3 frames per second with nearly a half-second in latency from raw to processed image. Since this Raspberry Pi has four cores, this was a massive waste of potential processing power and speed. Two solutions to this processing power waste were attempted and the second was adopted.

The first solution attempted to split the image processing algorithm up into multiple steps that could be run in parallel on different frames. The idea was to buffer a number of frames in parallel processing steps. For example, one frame could have its edges detected while the next frame could be cast to gray scale and blurred. Once one process was done modifying its current frame, it would block while attempting to pass the image to the next process in the modification algorithm. This process is shown in figure A.4.

This solution provided little improvement over the single-process algorithm. While the percentage CPU in usage increased from 25% to 70%, no increase the frame speed was noted. This resulted from the blocking nature of this algorithm and the large amount of data (an entire image buffer) that was passed between processes.

The second and final solution attempted was to run the the entire image processing algorithm in a single process, but have multiple processes running at the same time. One a frame from the camera was captured, it was passed to a pool of processes that would take a frame, completely process it, then pass the information back to the flight controller independently. With this solution, fewer images buffers needed to be passed between processes and there was no blocking. As a result, the frame rate of this algorithm increased from about 2 frames per second to 8 frames per second, a 300% increase! In addition, the latency decreased from 500 ms to 200 ms. An illustration of the algorithm can be seen below in figure 3.21

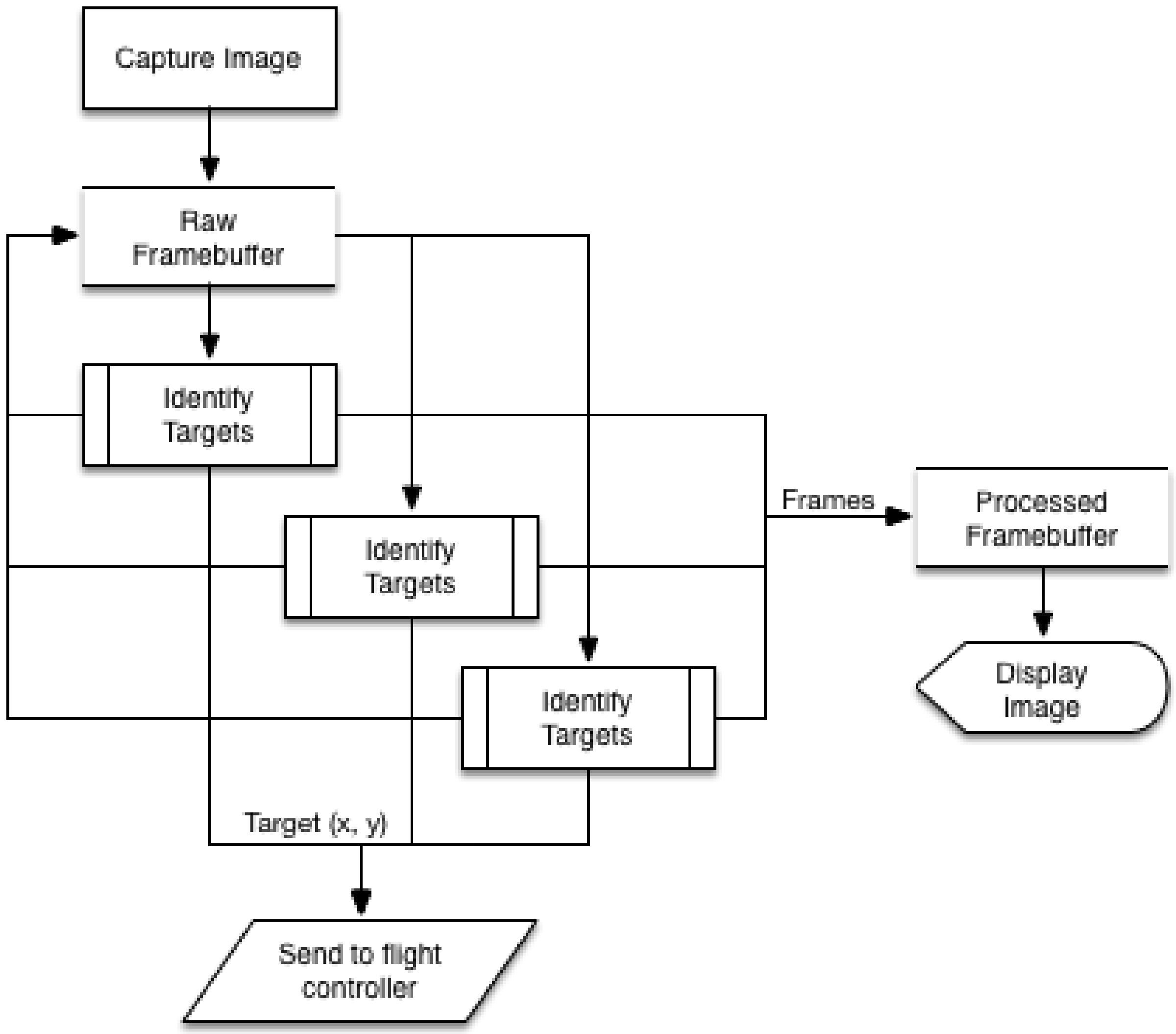


Figure 3.21: Image capture routine.

The actual limitations of this new parallel-processing algorithm lie in how fast the camera could capture individual frames and pass it to the main processing script. The Raspberry Pi camera simply could not take enough still-frame images to saturate the image processing algorithm. Despite this, the increase in frames per second and decrease in latency was sufficient for this project's purposes.

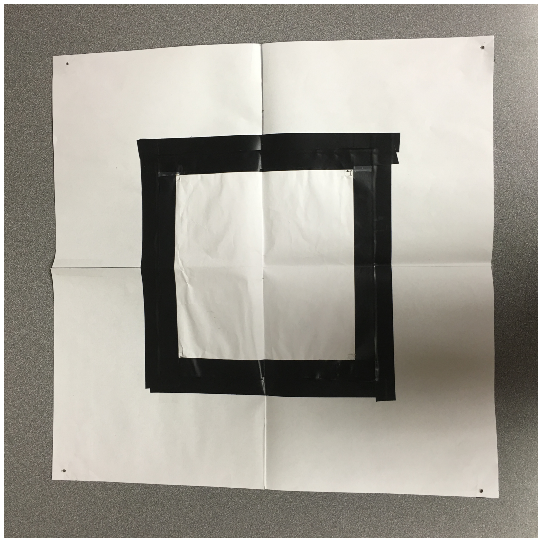
3.8.6 Image Streaming

The final component of the vision system was a reasonable method of monitoring the vision system and peeking in at what it could see. Initially VNC remote desktop was used to view the raw images being displayed by the vision algorithm, however with the weak Wi-Fi link up to the UAS platform it averaged 0.25FPS while viewing the stream, which was effectively unusable.

The solution was to publish the camera images to a web stream. This stream was served up as a web page by the Python code running on the Raspberry Pi using a basic MJPG stream. The streaming code retrieves the raw frames from the processing buffer and converts them to JPG, and then serves them to any connected clients via a web page. The code for this can be seen in Appendix B.1.

This code configures a listener to listen for any HTTP connection attempting to connect to the Raspberry Pi on port 8080 (not shown). When a connection is made, if the web page that the client is trying to access ends in .html, the second if statement will execute and serve an HTML page to the client that contains a basic MJPG stream URL. When the client browser loads the page, it will fetch the MJPG stream from the Raspberry Pi (first if statement). The while loop contained in this block will execute until the client disconnects. Effectively it repeatedly replaces the JPG shown in the browser window with a new one from the vision stream. This is a very basic method of streaming, but it performs well over the network connection that is available, yielding very low latency and high framerates.

Figure 3.22 shows the vision target itself, side by side with the target as seen by and identified by the camera. This snapshot of the identified target is pulled directly from the video stream.


(a) Vision target (unidentified).

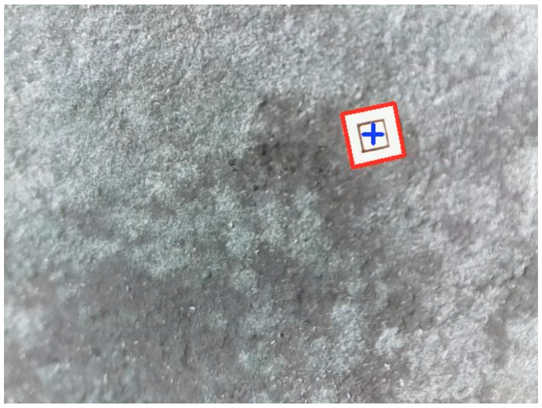

(b) Vision target, identified and marked by the vision algorithm.

Figure 3.22: Vision targets.

3.9 Drone Electronics

This section discusses the connections and setup of the electronics on the UAV.

One power module connects the battery to the power distribution board. This power module also has a 5V output to power the Pixhawk, radios, GPS Receiver and buzzer. At roughly 2.5 amps, this module does not have enough to power anything else. Browning out of several components was observed when additional pieces of electronics were added to this 5V system. In order to fix this, two additional 5V buck converters were purchased and attached to the power distribution board in parallel. Placing more than one of these power modules, or buck converters, in series resulted in destroyed components. This is most likely due to the current sense resistors in the power modules, as putting two in series would throw off the values and cause the "smart" power modules to use the wrong settings for the amperage in use. One of the additional two power modules was used to power the Raspberry Pi, as it can draw up to 2 amps. The other power module was was used to power an external USB hub for a high powered wireless adapter. All of the grounds in this system are tied together.

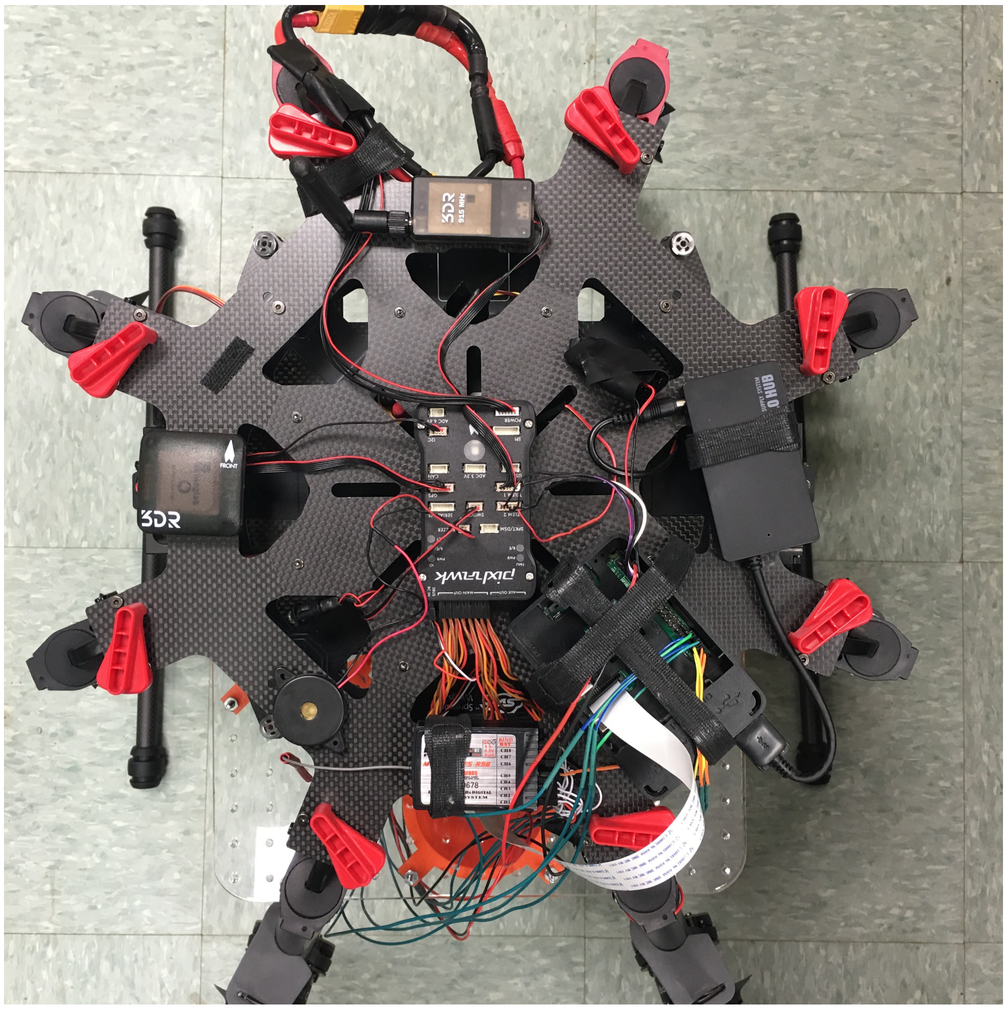


Figure 3.23: Top of the airframe, showing the layout of the various controller components.

The payload bay system needed a very small amount of current at 5V and up to 1.5 amps at 12V, 5V for running the chip logic and 12V for running the actual solenoids that hold the payload in place. 12V power was supplied by a buck converter attached in parallel with the two 5V buck converters mentioned above and supplied 2.2 amps to the solenoids. As each solenoid was limited to 1 amp by the H-bridge chip controlling it, this would allow us to open two solenoids at once if needed. The 5V source for the chip logic was connected directly to the Raspberry Pi as the h-bridge chips needed a very small amount of current at 5V. The trigger pins on the h-bridge chips were connected to GPIO pins on the Raspberry Pi to allow the Raspberry Pi to trigger the solenoids and drop the payloads. Due to the fact that the GPIO pins are floating when the Raspberry Pi first boots up, the script controlling the GPIO pins must be run before the h-bridge chips are connected to 12V power. The protoboard for the H-bridge chips can be seen in Figure 3.24.

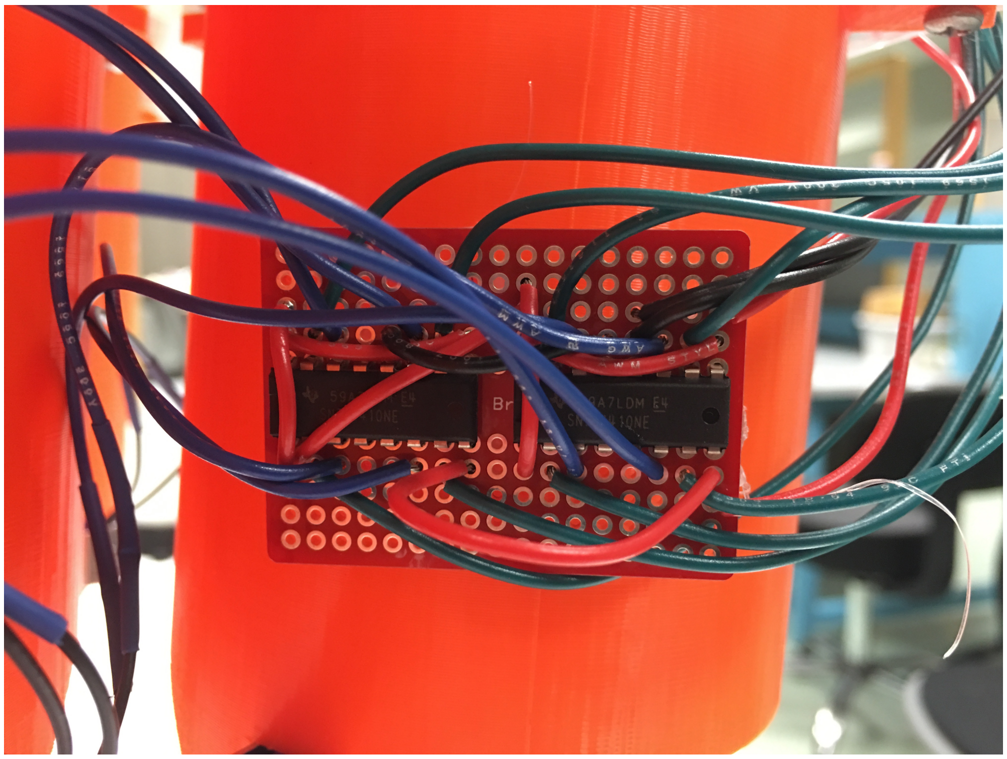


Figure 3.24: H-bridge chip board for controlling the solenoids.

3.10 System Integration

The penultimate stage in the project development was the system integration stage. Previously, a large number of discrete subsystems had been created each with their own set of features. In order to create a functional system these systems had to be integrated together.

3.10.1 System Organization

Figure 3.25 shows the flow of data between the different subsystems within the UAS, and the general organization of the different subsystems in relation to each other.

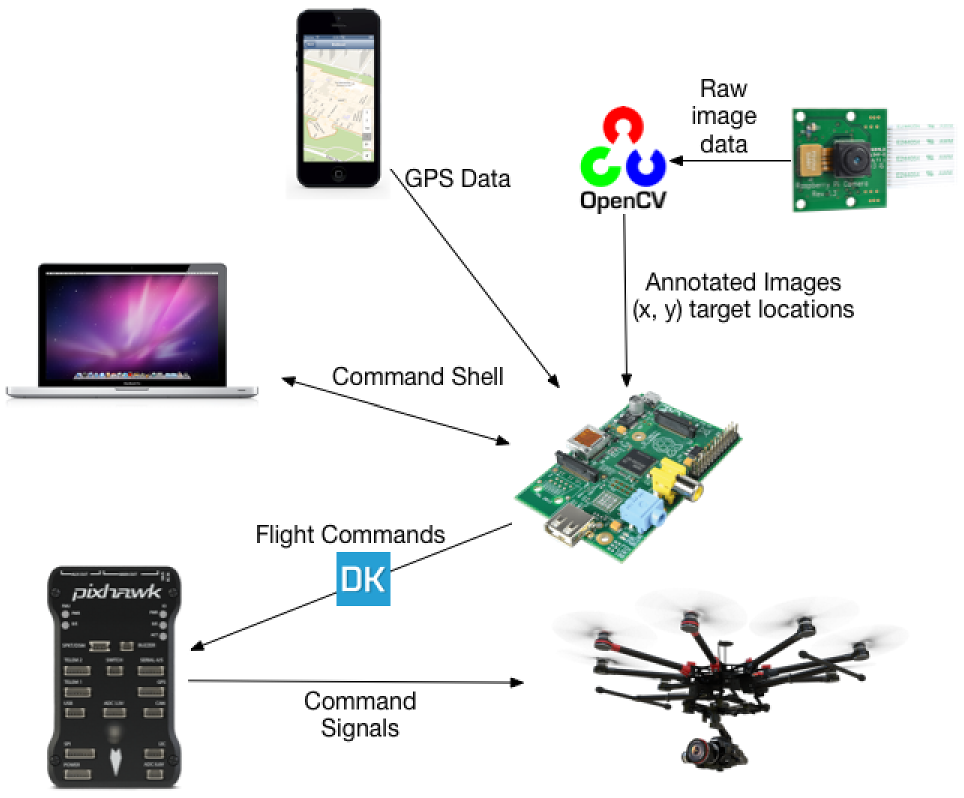


Figure 3.25: Overall system data and control flow.

The different subsystems are as follows:

- Terminal Based Command Shell
- Rasbperry Pi
- Raspberry Pi Camera
- OpenCV Vision Framework
- Pixhawk
- Airframe
- Smartphone-based GPS Data Collector

The Pixhawk and the airframe are the base layer of the whole control system. The Pixhawk is the only piece of hardware that can directly interface with and control the airframe. All flight commands need to be passed through the Pixhawk. The Raspberry Pi passes command signals to the Pixhawk using 3DR Dronekit. This framework allows the Raspberry Pi to send messages to the Pixhawk to tell it to perform various flight actions.

The Raspberry Pi is controlled through a basic command shell which runs over secure shell. The command shell allows the operator to command the Raspberry Pi to trigger different operation and flight modes. The control programs on the Raspberry Pi receive signals from the command shell as well as the onboard vision system and the GPS position input from a smartphone or GPS-enabled device. The vision system uses OpenCV on the Raspberry Pi, and receives images from the onboard Raspberry Pi camera.

3.10.2 Main Control Loop

To initialize the UAV and begin operation, the Raspberry Pi runs a few different initialization steps before entering a main control loop. It first opens up sockets to listen for image information from the vision subsystem, coordinates from the GPS system, and commands from the command shell. It then connects to the vehicle over the MAVLink/DroneKit connection, and waits for the vehicle to initialize. Once the Pixhawk reports that the airframe is ready, the UAV enters the main control loop. This progression can be seen in figure 3.26.

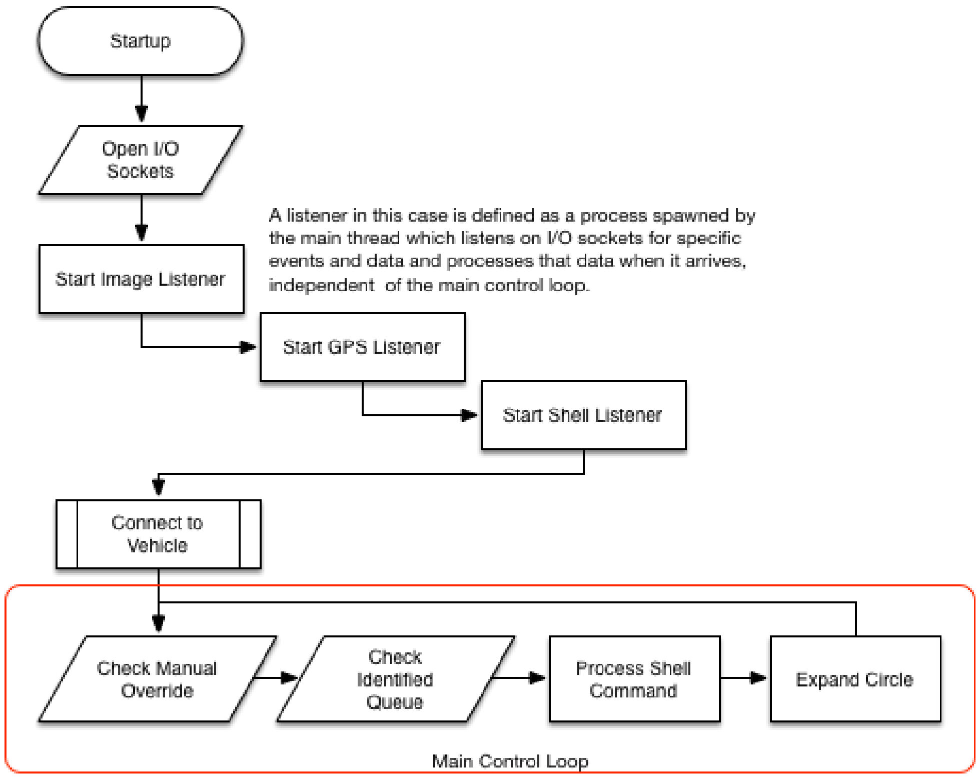

Startup
Open I/O Sockets
A listener in this case is defined as a process spawned by the main thread which listens on I/O sockets for specific events and data and processes that data when it arrives, independent of the main control loop.
Start Image Listener
Start GPS Listener
Start Shell Listener
Connect to Vehicle
Check Manual Override
Check Identified Queue
Process Shell Command
Expand Circle
Main Control Loop

Figure 3.26: Main control loop program flow.

The progression of the main loop is as follows. First, the program checks to see if the manual override switch has been enabled (seen in Figure A.6). If it has, it hands over control to the pilot. If not, it continues to the next mode and checks the image queue to see if the imaging system has found any targets (seen in Figure A.1). If it has, it sets a flag to start circling outwards to home in on the target. Next, it checks the command queue to see if the operator has entered any commands (seen in Figure A.2). If they have, it will process them. Finally, it increases the radius of the circle if there is a circle command running. This loops back around to the start.

As mentioned, the main control loop opens up sockets to listen for events from subprocesses. These include the image processing system, the command shell system, and the GPS system. Each of these is configured to run as a completely independent process so as to allow the main loop to handle flight navigation. The main loop needed to be completely non-blocking in order to allow the loop to check the manual control override. As it turned out, in order for the pilot to override the autonomous routine the main loop needed to not wait at all. If it blocked and was not in continuous communication with the Pixhawk, the Pixhawk would completely ignore the manual override and only complete the autonomous routines.

To achieve this, the main loop communicates with various subprocesses using non-blocking sockets and queues. Each of the subprocesses (imaging, shell, GPS) connects to the main control loop through a socket. These processes each generate data and interact with the operators and then pass the relevant information along to the main control loop. This ensures that the loop remains non-blocking and the manual override works under all conditions.

3.11 Drone Control Interface

This section will cover the interface used to control the drone, 3DR's Dronekit. As stated in the previous section, Dronekit is an open source framework that can be used to control the UAV. It provides bindings in the Python programming language to allow for complete control of the UAV. To set up Dronekit, a serial cable needed to be connected between the Raspberry Pi and the Pixhawk, and some basic libraries were installed on the Raspberry Pi to allow for it to use the newly-connected interface.

Dronekit provides a number of built-in functions to allow for controlling the drone. It allows for position control in terms of the GPS coordinates, velocity control, path planning, and takeoff/landing. It also allows users to define custom commands that extend the framework in ways that are not built in.

For this application, the first command used at the beginning of each flight is the takeoff command. This causes the drone to takeoff to a specified height and loiter there until given further commands. The code then commands the drone to move to a specified GPS coordinate which has been input by the user. It then begins searching for the vision target by moving in an expanding circular pattern. When the target is found, the drone moves to "home in" on the target by using a custom-defined relative motion algorithm explained in

Section 3.12.

3.12 Centering and Deployment

The final stage of the project was to develop the code that centered the UAV above the target and deployed the payload. Once the UAV identifies the target, it brakes and hovers. Simply seeing the target does not guarantee that the drone is directly above it. At an altitude of 6 meters, a camera field of view of 48.1 degrees, and a camera aspect ratio of 4:3, the UAV can see a horizontal distance of $2 * 6 * tan(48.1/2) = 5.3$ meters and a vertical distance of $5.3 * 3/4 = 3.975$ meters. The identified target can be anywhere within this approximate 4x5 meter area.

Centering over the target is done in two steps. First, the UAV determines where the target is, then it must navigate to that position. Smalls errors build up in this process due to imprecise measurements and calculations, so the process must be iterated until the UAV is within an acceptable tolerance above the target.

The UAV first determines where the target is in a Front-Right-Down (FRD) coordinate frame. If the target in the camera image is some number of pixels above and to the right of center, this indicates that the UAV must move forward and to the right. The distance to navigate in this frame of reference is determined by the relationship between the aspect ratio of the camera and the true viewing area. If the target is perfectly centered, it would be at a pixel position (320, 240). Thus, if the position of the target in the image is (x,y) the true distance from the target is determined by the following:

$$\frac{x-320}{640pixels} = \frac{Right}{5.3m} \qquad \frac{y-240}{480pixels} = \frac{Front}{3.975m}$$

Once this position had been determined in the FRD frame of reference, it needed to be transformed into a navigable frame of reference. The software navigation commands are all executed in a local North-East-Down (NED) frame of reference. This frame measures some distance North, East, and down in meters from the original takeoff location.

The conversion between the two frames is done in two steps. First, the FRD frame is converted into a differential NED frame. The FRD frame indicates that the drone needs to move some distance from its current location, not from some global frame, so the transformed frame should also be some differential distance from the current position. The transformation between the FRD frame and differential NED frame is accomplished using the following transformation matrix:

$$\begin{bmatrix} North \\ East \end{bmatrix} = \begin{bmatrix} cos(\theta) & sin(\theta) \\ -sin(\theta) & cos(\theta) \end{bmatrix} \begin{bmatrix} Front \\ Right \end{bmatrix}$$

where θ is the yaw angle from the magnetic north. As a note, the down component of the FRD frame was not transformed because it is identical to the down component of the NED frame.

The next step was to transform differential NED movements into a local NED position. This was accomplished by simply adding the differential NED movement to the current NED position. With the transformations complete, the UAV could identify a target and determine where it needed to go to center above the target.

The last step was iterating the identification and navigation process. While testing, it was noted that the UAV would approximately center above the target, but its position would be altered by the wind or inaccurate control maneuvers. To better center, the process was iteratively called until the UAV was within 0.5 meters of the target.

Chapter 4

Results

All systems of the UAV were proven to work individually, though all systems were never tested at once. The UAV system was able to:

1. Automatically takeoff
2. Wirelessly receive GPS data from a transmitter
3. Navigate to a GPS coordinate
4. Circle outwards slowly, searching for a target
5. Identify a target from 6 meters and stop
6. Center above the target
7. Drop a payload
8. Fly quickly away
9. Automatically land

The integration of these systems was never tested due to a crash on the final testing day. The UAV took a nose dive on takeoff and required repairs more substantial than what could be done in-house. That said, it seems like the system would have been able to function as intended. With further refinements and testing in a future project, the system could fulfill all of the requirements set out in the project description.

4.1 Flight Control System

The DroneKit-based flight control system worked almost as well as expected. The built-in standardized functions worked well (takeoff, land, go to GPS coordinate), however issues were encountered when trying to do anything custom (move the airframe one meter to the east of its current position). In developing these custom commands the team encountered many undocumented components of DroneKit which led to unexpected and dangerous flight characteristics. It was also discovered that when communicating with the UAV over DroneKit the Raspberry Pi would capture control of the UAV and not relinquish it if the code on the Pi was waiting for user input. Therefore a set of interconnecting processes were developed to make sure that the main process running the DroneKit connection was never waiting for input. This code for the main DroneKit process can be seen in Appendix B, Section B.5 and the user shell that commands the flight code can be seen in Section B.4.

4.2 Target Identification System

The target identification system worked reasonably well. Once the lens modifications were added to the Raspberry Pi camera, the camera was easily able to identify a target from 6 meters away. While testing the system outdoors, a number of things were noticed. Under certain lighting conditions, the UAV had trouble identifying the target due to reflectance issues off of the white paper and electrical tape. In addition, the constant motion of the UAV and slight vibrations made target identification less consistent than indoor, stationary tests. With these observations in mind, the qualifications for a 'square' target were lowered and the size of the target increased, greatly improving the performance of the identification algorithm outdoors. The rest of the system worked perfectly. The camera images were streamed seamlessly to a webserver and the centers of identified targets were passed asynchronously to the flight control code. This code can be seen in Appendix B, Section B.6.

4.3 Payload Delivery System

The payload delivery system worked very well, with a few minor hiccups. The payloads were cheap and easy to create (in line with the goal for cheap and accessible parts), easy to load into the system, and deployed with enough impact force to detonate the targeted landmine. While testing, it was noted that the payload would occasionally get stuck while deploying. This was believed to result from them getting stuck on the solenoid pins on the way out of the tubes. While an occasional stick was noted, the vast majority of the time the payloads were delivered perfectly. Payload delivery was controlled through GPIO (General Purpose Input / Output) pins on the Raspberry Pi using an H-Bridge solenoid driver circuit. The driver circuit and GPIO pin combination worked perfectly every time.

4.4 GPS Data System

The GPS data system worked perfectly software-wise, but experienced issues with the actual GPS data. When a smartphone connected to the GPS webserver run on the Raspberry Pi, the phone would pass its GPS location as data to the server as if it were the mine detection robot flagging a mine. If this data was correct, the UAV would autonomously navigate to the location of the phone. Unfortunately, it was discovered that this GPS data was oftentimes not correct. Most smartphones cache the last wifi-determined GPS coordinate and would access that instead of pull it from cellular data. Since no system was connected to the internet, there was no way to verify if a GPS coordinate was correct. Since the system was proven to work, a known and verified GPS location was logged and passed to the UAV through the terminal instead. This solution worked for the remainder of the project. The GPS code can be seen in Appendix B, Section B.7.

Chapter 5

Discussion

5.1 Airframe

The airframe itself is very easy to work with. It is well designed and assembly was simple. While not out of the box compatible with the Pixhawk flight controller system, no major modifications were needed to make it work.

The only part of working with this airframe that was less than ideal was getting replacement parts. As sending the airframe back to DJI involved a turn-around time of 1-3 weeks, this was not acceptable and alternatives had to be found. The first alternative was a store that shipped parts from China, and after issues with shipping time another alternative had to be found. Another seller in the US was found, and the shipping time was dramatically reduced. This proved to be the best alternative.

Repairing the airframe was extremely easy. The entire frame was designed with engineered failure clips on the arms and foldable propellers. When a crash occurred, the plastic clips would break and the propellers would wold, protecting the carbon fiber are and center frame from extreme stresses. These clips and propellers were cheap to replace and very easy to reinstall.

After a particularly bad crash in which more than the clips and propellers broke, the airframe was returned to DJI for servicing. As this happened at the end of the project and the lead time was 2-3 weeks, there is nothing to report on the experience with DJI support as of yet.

With respect to protecting the electronics from shrapnel, the airframe itself keeps most of the sensitive electronics out of the path of the shrapnel. All of the radios and flight controller electronics have been mounted on the top of the airframe, which does expose them to the weather more, but it also provides much better protection from shrapnel.

As for the actual weather, light very snow or mist are the maximum acceptable precipitation that can be flown in. This is because all of the electronics are exposed to the elements. The radios and flight controller board can be covered in plastic wrap or otherwise protected, but the ESCs themselves are mounted directly under the motors and have a fan and vents to help with cooling. This means that flying in any serious rain or snow could damage them and is not recommended. As for wind, a max wind speed of 5-10 mph is the maximum that is safe to fly in. The control system has no problem compensating for steady winds of these speeds and gusts of around 15-20 mph, but in order for the system to accurately aim at the target it should be used in as little wind as possible.

5.2 Payload Bay Prototypes

The final payload bay worked well. During testing no early or unplanned releases happened. The payload bay doors opened every time the Raspberry Pi triggered them. However, not every payload dropped when the door was opened.

This is due to the fact that the solenoid pin sticks out into the area that the payload falls through on release. A small washer on the solenoid pin caught on the plastic of the payload and kept it from falling any farther. As the payload would always fall before the drone would touch down this was never verified to be the cause of the issue. However, for this to happen the washer on the solenoid pin would have to catch the bag, and therefor the bag would most likely have to tear in order for the payload to fall. When the payloads were recovered from the test flights, the payload that had failed to fall immediately was the first one to break, so this is the most likely case.

While the final payload bay has the payload tubes arranged in a t shape, no significant changes in flight characteristics were noticed before and after the payloads are deployed. When individual payloads are deployed the UAS does not shoot upwards or off to the side. The control system handles the change in center of mass, and the overall change in mass, with no issues.

5.3 Drone Control Interface

During software development, it was quickly noted that a drone control hierarchy needed to be worked into the control code. Originally, the user shell was not run in a separate process on the raspberry pi, rather in the main control loop. During the shell operation, the python command *raw_input*() was called, blocking the process until a user input some sort of data through a terminal.

Initially, this blocking action was not a problem. The goal was to have to UAV accomplish one step of the overall mission at a time and wait for a terminal user to specify the next task. This way, not every function needed to be tested at once and the terminal user could stop the drone's action at any time.

It was discovered, though, that this blocking step prevented the pilot from overriding the raspberry pi's commands. This was both a safety and a legal concern. If the wifi connection between the raspberry pi and a user's computer dropped, a user would have no way of controlling the UAV, and the pilot would have no way of taking back control. The UAV would be stuck in the air. The In addition, U.S. law requires that all autonomous UAV must have immediate override switches. At the time, the cause of the problem was unknown, so 3DR support was contacted.

After contacting 3DR support (the manufacturer and maintainer of the Pixhawk Flight Controller and the DroneKit software interface), a software engineer informed the team that the main control script could block at no time during flight operation. Unfortunately, this property was not specified anywhere in the DroneKit documentation or example scripts. A further discussion of the DroneKit interface is included in the next section.

The entire structure of the software (but mostly the control structure) needed to be reworked. As mentioned in previous sections, the shell, image processing, image streaming, and gps coordinate functions were all transferred to separate scripts and run on individual processes. While this allowed for the functions to run in parallel, it also increased the complexity of the code, as data then needed to be transferred asynchronously between the processes. To accomplish this, inter-process queues were created to manage the data transfer.

With the control script no longer blocking, one final piece of code needed to be added. The 3DR engineer suggested that the main control loop poll the controller values constantly, checking to see if a manual override switch was thrown. If thrown, the software should change the UAV mode to 'Loiter', a mode that gave the pilot full control. After implementing and testing this software, it was noted that it was not actually needed. As long as the main control script is not blocking, the Pixhawk flight controller will automatically detect when a pilot is taking control and will yield control.

5.4 DroneKit & Mavlink Interfaces

DroneKit acts as a software abstraction layer that communicates to the Pixhawk flight controller through custom MavLink messages. The purpose of DroneKit is to supply an easy-to-use library to control UAV's for simple purposes such as simple GPS flight paths or gimbal/camera control maneuvers. When DroneKit does not supply a software command for a desired maneuver, the use of custom MavLink messages is required. Any aerial maneuver should be able to be accomplished through one of these two interfaces.

Unfortunately, the documentation state of these two interfaces is rather poor as they are both still in development. For example, as mentioned in the previous section, if a DroneKit control script blocks at any time, a pilot cannot assume manual control. Despite the safety and legal importance of this specification, it is mentioned nowhere in the documentation.

The API reference itself, *http://python.dronekit.io/automodule.html*, is a text wall, filled with circular references. The information contained, while generally helpful, more than once left the team in a rut. For example, the documentation links to ArduCopter copter modes as the supported flight modes, but while testing, it was discovered that a number of these modes, such as Altitude Hold, simply are not supported. This particular discovery was made when the software tried to change the flight mode to altitude hold and crashed instead, leaving the terminal user without any control.

A final example is the lack of support of certain frames of reference. When writing the centering routine, a body frame of reference was desired. A body frame of reference would allow the software to command the craft to go North or East by 2 meters from its current location. Indeed MavLink includes this a frame ($MAV_FRAME_BODY_NED$), but when it was implemented into the software, it behaved erratically and inconsistently. Due to the poor documentation of this frame of reference and the interactions between DroneKit, MavLink, and the PixHawk flight controller, the team could not determine the cause of this error, so this frame of reference was abandoned.]

It is understood that these software implementations are still very much works in progress. They have great strengths such as their great support of GPS and waypoint navigation. While not supported in default, precise position and velocity control can be attained, though beware the poorly documented and possibly unsupported commands.

Chapter 6

Conclusion

Landmines have been banned by the international community for nearly two decades now, yet they are still the cause of thousands of civilian deaths every year. Hundreds of thousands of mines are still buried around the world, particularly in the Middle East, Southeast Asia, and Latin America, remnants of wars generations past. Efforts to clear these landmines have stalled. The costs of machinery, manpower, and training are too high for the impacted regions to purchase. A new solution needed to be created.

The purpose of this MQP was to create a lower-cost alternative to landmine clearing systems that utilizes recent advances in UAV technology. A UAV system was created that can autonomously navigate to landmine locations, search until a tagged landmine has been found, drop a payload from 6 meters upon a landmine, detonating it, and escape with little to no damage. The system developed for this project was able to verify that many parts of the system are functional, albeit not yet as a whole. This system, while not ready for commercialization, has proved that this task is in fact doable, and is a viable alternative to the current expensive solutions.

Chapter 7

Future Work

7.1 Mine Detection Robot

An implied portion of this project is the mine detection robot. This robot would go along with the UAV, and provide a means of finding and marking the landmines in the field. Without this robot, marking the mines would require human effort, endangering lives. There are a few requirements for such a robot to allow it to integrate well with the drone platform, namely:

1. Wi-Fi connection to allow the detection robot to relay to the drone where the mines are through the GPS script
2. GPS receiver to be able to gather information about the mine's approximate location
3. System for placing the visual flags on the land mines without detonating them
4. Ability to find land mines that are partially obscured by brush or small amounts of dirt

By following these requirements, the mine detection robot should be able to integrate with the UAV seamlessly. The mine detection robot can be constructed in any manner so long as it meets these requirements, leaving it as a perfect candidate for a future MQP. The system constructed for this project allows for easy extensibility, making it easy for the detection robot to integrate with it.

7.2 Mine Detonation Verification

A system not integrated into the current design is a system to verify whether or not a landmine had been successfully detonated or not. As one could imagine, tagging a nondetonated landmine as detonated or safe would result in safety concerns. A number of ideas for a system were discussed, though never implemented or tested because they were outside the scope of the project.

Visual confirmation may be difficult due to the shrapnel constraint of the project. The UAV does not have time to hover above the landmine and watch to see if the mine has been detonated. It must immediately accelerate horizontally to escape any possible shrapnel. A possible solution is to fly back to the drop location and see whether or not the target is still present. If a target is still present, this indicates the mine has not been detonated. A problem, however, could occur if the target was covered up in some way by the payload, resulting in a false positive for detonation.

Another possible solution could be sound detection. When a landmine detonates, it releases a large shockwave (soundwave). A small microphone on the UAV could detect such a soundwave and confirm that the mine has been detonated. This system does not require that the UAV return to the drop location, nor does it involve to possible false-positive problem.

7.3 Further Target Development

The target for the project was chosen rather arbitrarily. A distinct, easily identifiable target was desired, however a square was chosen simply because of readily available code that could already identify squares. There is plenty of room for further development and research into possible targets that are even easier to identify from 6 meters away, in different lighting conditions, or that would require fewer steps in the image processing algorithm.

7.4 Centering Algorithm

The centering routine was the final piece of software developed, and as such it had the least amount of time to be tested. One consideration that was never implemented in code was the offset of the payload bays from the camera. If a target is centered, this indicated that the target is directly below the camera, not the payload bays. When a payload is released from that position, under ideal conditions the payload will not hit the target, rather some offset distance away from the target. In future projects, this offset needs to be taken into account. In addition, the offset from each one of the payload bays needs to be considered; they are all different distance away from the camera.

7.5 Payload Bay

A new set of payload tubes should be designed that are more likely to release the payloads without snagging. This could be done in one of several ways. For example, the solenoids could be placed on the doors instead of the outside tube. This way there would be no part sticking out for the payloads to snag on. Alternatively, a lip could be designed to block the payload from hitting the solenoid on the way down.

Originally, the payload tubes were designed with PVC pipes. This solution proved to be more work and manufacturing labor, but the parts were much more readily available than 3D printed custom parts. The PVC pipes were discarded in favor of less labor, though they were never proven to be unusable. This solution could be pursued in line with the goal of maintaining cheap and easily-accessible parts.

References

3DR. (2016a). *3dr pixhawk - 3dr.* Retrieved from https://store.3dr.com/products/3dr-pixhawk

3DR. (2016b). *Dronekit by 3d robotics.* Retrieved from http://dronekit.io

DJI. (2016). *Flight controllers — dji.* Retrieved from http://www.dji.com/products/flight-controllers?www=v1

Force, U. A. (2014). *Official u.s. government information about the global positioning system (gps) and related topics.* Retrieved from http://www.gps.gov/systems/gps/performance/accuracy/

Group, L. M. C. (2004). *Toward a Mine-Free World Executive Summary* (Tech. Rep. No. October). Retrieved from http://www.the-monitor.org/media/1759694/lm2004execsum-txt.pdf

Landmine and ied rollers - hri. (n.d.). HRI. Retrieved from http://humanisticrobotics.com/products/landmine-and-ied-rollers/

Mine flail. (n.d.). Wikimedia Foundation. Retrieved from https://en.wikipedia.org/wiki/mine_flail

Monitoring, I., & Committee, R. (2014). *Landmine monitor 2014* (Tech. Rep.). doi: 10.1017/CBO9781107415324.004

Pmn-mine. (n.d.-a). Wikimedia Foundation. Retrieved from https://de.wikipedia.org/wiki/pmn-mine

Pmn mine. (n.d.-b). Wikimedia Foundation. Retrieved from https://en.wikipedia.org/wiki/pmn_mine

Rodriguez, J., Castiblanco, C., Mondragon, I., & Colorado, J. (2014). Low-cost quadrotor applied for visual detection of landmine-like objects. *2014 International Conference on Unmanned Aircraft Systems (ICUAS)*, 83–88. Retrieved from http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6842242 doi: 10.1109/ICUAS.2014.6842242

Soviet / russia pmn-1 bakelite landmine. (n.d.). Retrieved from https://www.buymilsurp.com/soviet-russia-pmn1-bakelite-landmine-p-5098.html

Swinton, R., & Bergeron, D. (2004). Evaluation of a silent killer, the pmn anti-personnel blast mine.

Youngblood IV, N. E. (2002). The development of landmine warfare.

Appendix A

System Flow Diagrams

This appendix contains many secondary flow diagrams which explain in more detail various parts of the UAS system.

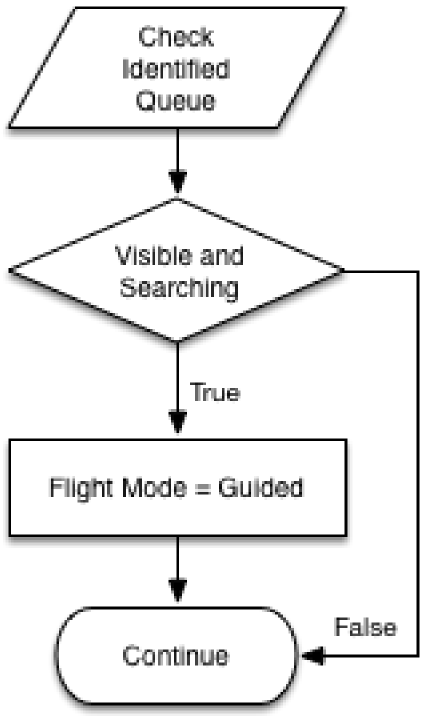


Figure A.1: Check image processing queue flow diagram.

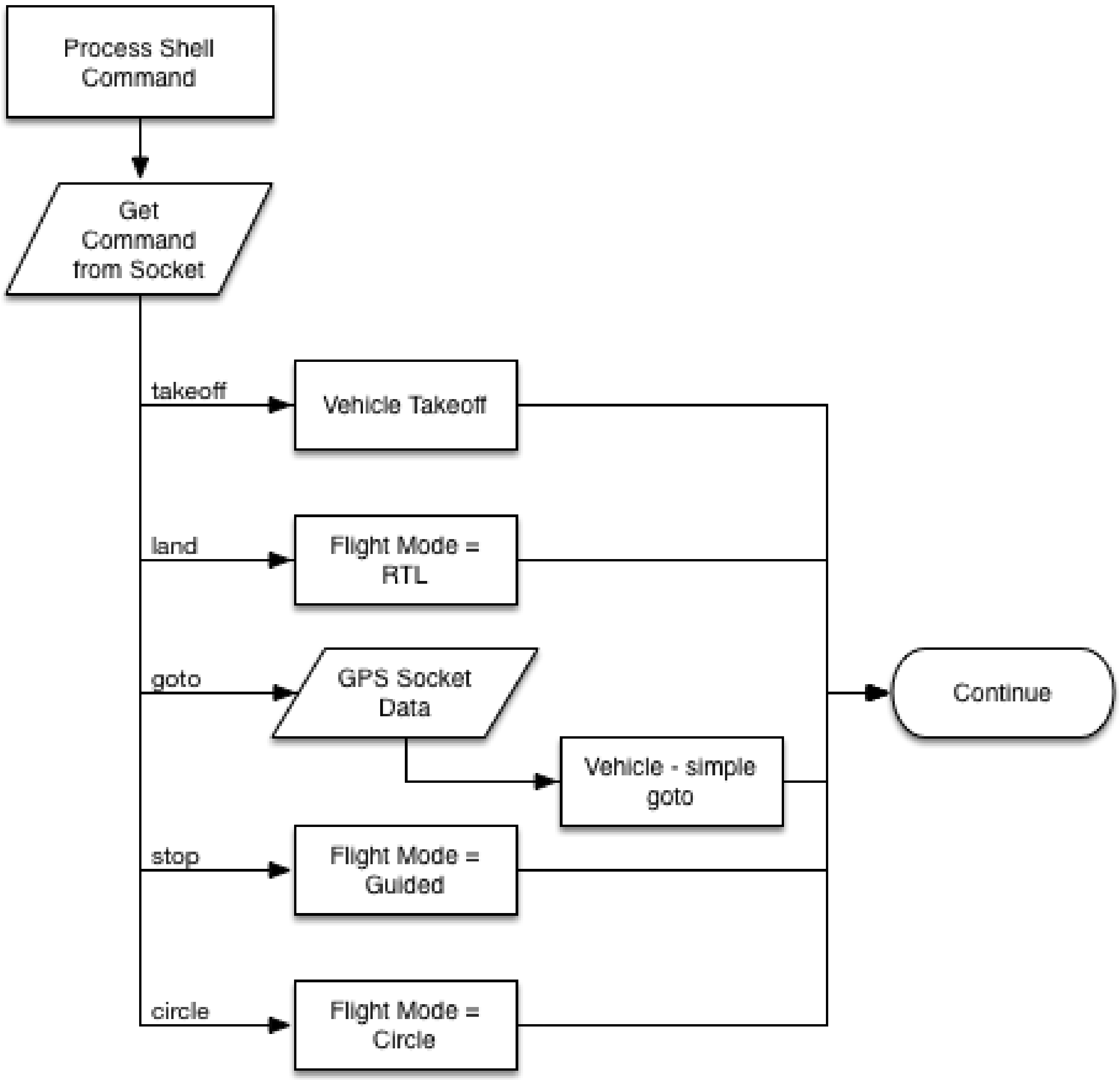


Figure A.2: Shell command processing flow diagram.

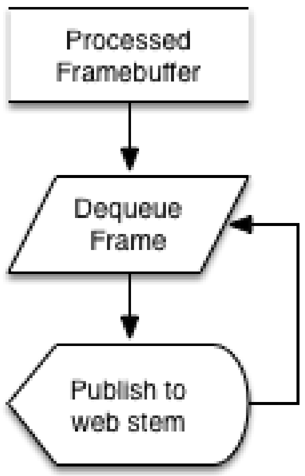


Figure A.3: Image display process flow diagram.

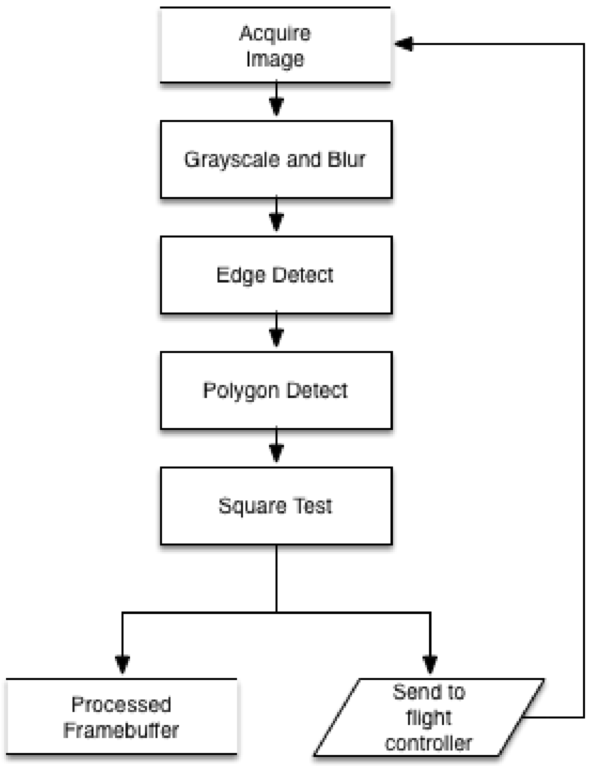


Figure A.4: Old target identification algorithm process diagram.

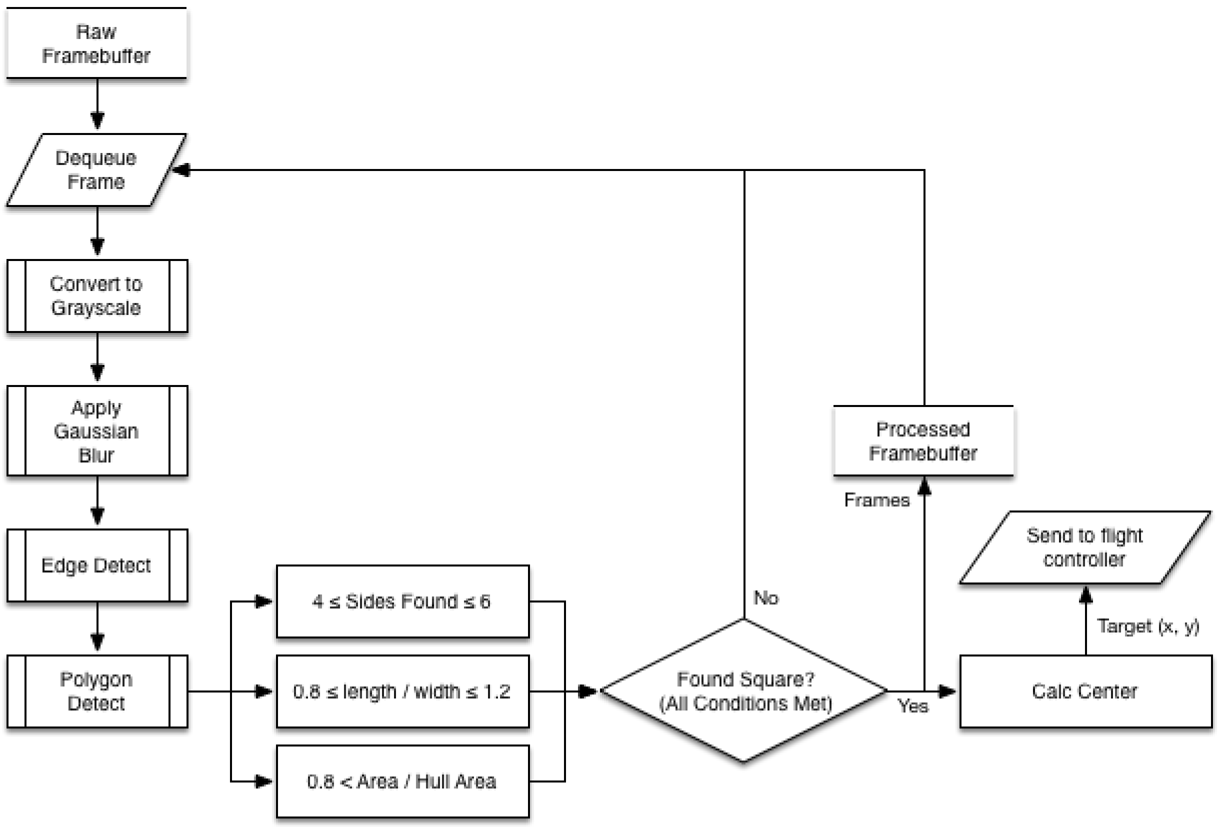


Figure A.5: Target identification algorithm process diagram.

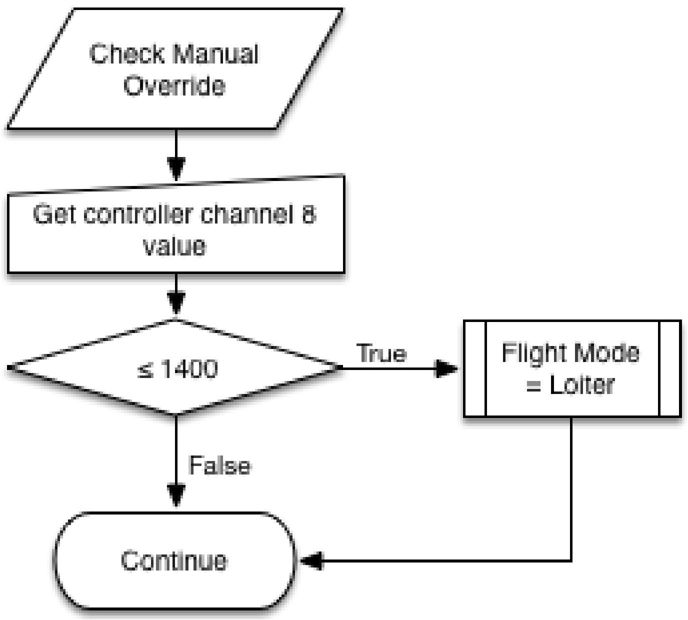


Figure A.6: Manual override check process flow diagram.

Appendix B

Source Code

This appendix provides examples of source code that was created as part of the project implementation.

B.1 Python Image Streaming Code

```python
class CamHandler(BaseHTTPRequestHandler):
    def do_GET(self):
        if self.path.endswith('.mjpg'):
            self.send_response(200)
            self.send_header('Content-type','multipart/x-mixed-replace; boundary=--jpgboundary')
            self.end_headers()
            while True:
                try:
                    jpg = Image.fromarray(final_queue.get(block=True, timeout=None), 'RGB')
                    tmpFile = StringIO.StringIO()
                    jpg.save(tmpFile,'JPEG')
                    self.wfile.write("--jpgboundary")
                    self.send_header('Content-type','image/jpeg')
                    self.send_header('Content-length',str(tmpFile.len))
                    self.end_headers()
                    jpg.save(self.wfile,'JPEG')
                except KeyboardInterrupt:
                    break
            return
        if self.path.endswith('.html'):
            self.send_response(200)
            self.send_header('Content-type','text/html')
            self.end_headers()
            self.wfile.write('<html><head></head><body>')
            self.wfile.write('<img src="cam.mjpg"/>')
            self.wfile.write('</body></html>')
            return
```

Listing B.1: Python image streaming code

B.2 MATLAB Code

```
function A = plotMaxDistance(mass, coefficient , area)

angles = [pi/4:pi/180:pi/2];


velocity = [];
distance = [];
time = [];
angls = [];

for angle = angles
    [d, v, t] = plotShrapnel(angle, mass, coefficient , area);
    temp = angle * ones(size(t));
    velocity = [velocity;v];
    distance = [distance;d];
    time = [time; t];
    angls = [angls;temp];
end
#{
figure(3);
hold on;
scatter3(0,0,0);
close all;
pause(2);
scatter3(angls, time, distance, [], floor(distance));
xlabel("Initial Ejection Angle in radians");
ylabel("Time in seconds");
zlabel("Total Distance in meters");
title("Ejection distance vs initial angle and time");
% print -djpg "3DPlotDistance.jpg"
#}

figure(4);
hold on;
scatter3(0,0,0);
close all;
```

```
pause(2);
scatter3(angls, time, velocity, [], floor(velocity));
xlabel("Initial Ejection Angle in radians");
ylabel("Time in seconds");
zlabel("Magnitude of velocity in m/s");
title("Ejection velocity vs initial angle and time");
%print -djpg "3DPlotVelocity.jpg"


% scatter(angles, time, distance);
% xlabel("Angle in Radians");
% ylabel("Maximum Distance in m");
% title("Maximum Distance vs Initial Angle of Shrapnel");
% print -djpg "distanceVAngle.jpg" -F:15

end
```

Listing B.2: Shrapnel maximum distance plotting code

```
function [r v t] = plotShrapnel(angle, mass, coefficient , area)
A = area; % 0.000025;       Cross sectional area of cube. (5mm)^2
p = 1.275;      %           Density of air. kg/m^3
C = coefficient; % 1.05;    Coefficient of drag
m = mass; % 0.0002;         Mass. kg
g = 9.81;       %           Gravity. m/s^2
V = 900;        % Initial velocity. m/s
theta = angle;  % Initial angle from horizontal

Vx = V*cos(theta);
Vy = V*sin(theta);
X = 0;
Y = 0;

dt = 0.001;%0.0001;
time = 5;

X_ = zeros(time/dt, 1);
Y_ = zeros(time/dt, 1);
Vx_ = zeros(time/dt, 1);
Vy_ = zeros(time/dt, 1);
theta_ = zeros(time/dt, 1);

counter = 1;

for t = [0:dt:time]

    X_(counter) = X;
    Y_(counter) = Y;
    Vx_(counter) = Vx;
    Vy_(counter) = Vy;
    theta_(counter) = theta;

    V2 = Vx^2 + Vy^2;
    Vx -= ((1/2) * p * C * A * V2 * cos(theta))/m * dt;
    Vy -= ((1/2) * p * C * A * V2 * sin(theta) + m*g)/m * dt;
    X += Vx * dt;
    Y += Vy * dt;
    if(Y <= 0)
```

```
        break;
    endif

    theta = atan2(Vy, Vx);
    counter++;

end

r = sqrt(X_ .^2 + Y_.^2);
v = sqrt(Vx_.^2 + Vy_.^2);
t = [0:dt:time]';

#{
figure(1);
hold on;
plot([0:dt:time], Vy_, 'b');
plot([0:dt:time], Vx_, 'r');
xlabel("Time in seconds");
ylabel("Velocity in m/s");
title("Velocity vs Time of Shrapnel");
legend("Vertical Velocity", "Horizontal Velocity");
print -djpg "shrapnelVelocity.jpg" -F:15

figure(2);
hold on;
plot([0:dt:time], Y_, 'b');
plot([0:dt:time], X_, 'r');
plot([0:dt:time], r, 'g');
xlabel("Time in seconds");
ylabel("Distance in m");
title("Distance vs Time of Shrapnel");
legend("Vertical Distance", "Horizontal Distance", "Radial Distance");
print -djpg "shrapnelDistance.jpg" -F:15
#}

end
```

Listing B.3: Shrapnel maximum velocity plotting code

B.3 Data Simulator

```python
import socket, pickle


def main():
    image_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    gps_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

    while True:
        try:
            gps_socket.connect(('localhost', 5002))
            break
        except:
            pass

    while True:
        try:
            image_socket.connect(('localhost', 5001))
            break
        except:
            pass

    while True:

        print "\n----------------------------------------------------------------------------------------------------\n"
        command = raw_input("What type of data?\t")

        if command == "gps":
            try:
                lat = float(raw_input("Latitude:\t"))
                lon = float(raw_input("Longitude:\t"))
            except:
                continue

            data = (lat, lon)

            serialized_data = pickle.dumps(data)
            gps_socket.send(serialized_data)

        elif command == "image":
            try:
                X = float(raw_input("X:\t"))
                Y = float(raw_input("Y:\t"))
            except:
                continue

            data = (X, Y)

            if X == -1.0:
                data = -1

            print data
            serialized_data = pickle.dumps(data)
            image_socket.send(serialized_data)

        else:
            print "Not valid data"


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

Listing B.4: Data simulator code

B.4 Shell Code

```python
#!/usr/local/bin/python

import socket, pickle


def main():

    client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

    while True:
        try:
            client_socket.connect(('localhost', 5003))
            break
        except:
            pass

    while True:

        print "\n--------------------------------------------------------------------------------\n"
        print "Options: takeoff, land, end, stop, circle, goto, ignore, search, clearq, print, override, drop, pos (F R D), yaw (angle), center"

        command = raw_input("What shall I do next?\n")
        data = pickle.dumps(command)
        client_socket.send(data)



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

Listing B.5: Shell code

B.5 DroneKit Interface Code

```
#!/usr/local/bin/python

from dronekit import connect, VehicleMode, LocationGlobalRelative
from pymavlink import mavutil
import time
import argparse
import sys
import pickle
from multiprocessing import Queue, Process, Value
import socket
import os
import math
import copy

# Global Variables and Flags
vehicle = None
ignore_target = True
tangential_speed = 50 # cm/s
circle_period = sys.maxint
home_location = None
last_centering_time = 0

# Process shared flags
identified = None
shutdown = None

# Sockets
image_socket = None
gps_socket = None
shell_socket = None

# Shared queues
image_data = Queue(maxsize=1)
gps_coordinates = Queue()
shell_commands = Queue()
last_image_location = Queue(maxsize=1)
```

```

# Simulator flag
SIM = False

def setup():
    global vehicle

    # Connect to the Vehicle
    print ”Connecting to the vehicle...”
    if SIM == False:
        vehicle = connect('/dev/ttyAMA0', baud=57600, wait_ready=True)
    else:
        vehicle = connect('tcp:localhost:5760', baud=57600, wait_ready=True)

    # Initialize the vehicle
    while not vehicle.is_armable:
        print ”Waiting for vehicle to initialize...”
        time.sleep(1)

    # Arm the vehicle
    # arm()

    print ”Set default/target airspeed to 3”
    vehicle.airspeed = 3


def arm():
    global vehicle

    print ”Arming motors”
    # Copter should arm in GUIDED mode
    vehicle.mode = VehicleMode(”GUIDED”)
    vehicle.armed = True

    while not vehicle.armed:
        print ” Waiting for arming...”
        time.sleep(1)


```

```
def takeoff(atargetaltitude=10):
    global vehicle, home_location
    home_location = copy.deepcopy(vehicle.location.global_frame)
    vehicle.home_location=vehicle.location.global_frame

    print "arming"
    # Arm the UAV
    arm()

    vehicle.mode = VehicleMode("GUIDED")

    print "Taking off!"
    vehicle.simple_takeoff(atargetaltitude)  # Take off to target altitude

    # Wait until the vehicle reaches a safe height
    while True:
        print " Altitude: ", vehicle.location.global_relative_frame.alt
        # Break and return from function just below target altitude.
        if vehicle.location.global_relative_frame.alt >= atargetaltitude * 0.95:
            print "Reached target altitude"
            break
        time.sleep(1)



def return_to_launch():
    global vehicle

    print "Returning to Launch"
    vehicle.mode = VehicleMode("RTL")

    # Wait until the vehicle lands to process the next command
    # while vehicle.location.global_relative_frame.alt >= 0:
    #   time.sleep(1)


def end_flight():
    global vehicle, shutdown

    # Disarm the vehicle
    vehicle.armed = False
    while vehicle.armed == True:
        print " Waiting for disarming..."
        time.sleep(1)

    # Close vehicle object before exiting script
    print "Close vehicle object"
    vehicle.close()

    # Shutdown the other processes
    with shutdown.get_lock():
        shutdown.value = 1


def go_to_coordinate(latitude, longitude, altitude=6, speed=1):
    global vehicle

    print "Navigating to point"
    vehicle.mode = VehicleMode("GUIDED")
    print latitude
    print longitude
    print type(latitude)
    print type(longitude)
    point = LocationGlobalRelative(latitude, longitude, altitude)
    vehicle.simple_goto(point, groundspeed=speed)

    # sleep so we can see the change in map
    # time.sleep(30)


def circle_POI():
    global vehicle, ignore_target, tangential_speed, circle_period

    # Serach for the target
    ignore_target = False

    # The circle radius in cm. Max 10000
    # The tangential speed is 50 cm/s
    speed = tangential_speed

    # Radius has to be increments of 100 cm and rate has to be in increments of 1 degree
    radius = int(100)
    period = 2*math.pi*radius / speed
    rate = int(360.0/period)

    vehicle.parameters["CIRCLE_RADIUS"] = radius
    vehicle.parameters["CIRCLE_RATE"] = rate

    vehicle.mode = VehicleMode("CIRCLE")

    # Update the global variable for the next circle
    circle_period = period


def update_circle_params():
    global vehicle, tangential_speed, circle_period

    current_radius = vehicle.parameters["CIRCLE_RADIUS"]
    current_rate = vehicle.parameters["CIRCLE_RATE"]

    new_radius = current_radius + 100
    new_period = 2*math.pi*new_radius / tangential_speed
    new_rate = int(360.0/new_period)

    vehicle.parameters["CIRCLE_RADIUS"] = new_radius
    vehicle.parameters["CIRCLE_RATE"] = new_rate

    # Update the global variable for the next circle
    circle_period = new_period


def stop():
    global vehicle

    vehicle.mode = VehicleMode("GUIDED")
    print "stopped"

```

```python
def clearGPSQueue():
    global gps_coordinates

    # Deletes and reinstatiates the GPS queue
    del gps_coordinates
    gps_coordinates = Queue()

def check_user_control():
    global vehicle, SIM

    if SIM == True:
        return False

    value = vehicle.channels['8']

    # print value

    if value < 1450:
        return True
    else:
        return False

def printData():
    global vehicle, gps_coordinates, home_location

    print "Alt: ", vehicle.location.global_relative_frame.alt
    print "Lat: ", vehicle.location.global_relative_frame.lat
    print "Lon: ", vehicle.location.global_relative_frame.lon
    print "Mode: ", vehicle.mode
    print "Ignore: ", ignore_target
    print "Distance from home: ", get_distance_metres(home_location, vehicle.location.global_frame)
    print "N E D: ", vehicle.location.local_frame.north, vehicle.location.local_frame.east, vehicle.location.local_frame.down
    print "yaw: ", vehicle.attitude.yaw

def drop():
    os.system('python ../experiments/drop_gpio.py')

# http://python.dronekit.io/guide/copter/guided_mode.html
```

```python
def get_distance_metres(aLocation1, aLocation2):
    """
    Returns the ground distance in metres between two `LocationGlobal` or `LocationGlobalRelative` objects.

    This method is an approximation, and will not be accurate over large distances and close to the
    earth's poles. It comes from the ArduPilot test code:
    https://github.com/diydrones/ardupilot/blob/master/Tools/autotest/common.py
    """
    dlat = aLocation2.lat - aLocation1.lat
    dlong = aLocation2.lon - aLocation1.lon
    return math.sqrt((dlat*dlat) + (dlong*dlong)) * 1.113195e5

def get_distance_linear(aLocation1, aLocation2):
    """
    Returns the ground distance in metres between two `LocationGlobal` or `LocationGlobalRelative` objects.

    This method is an approximation, and will not be accurate over large distances and close to the
    earth's poles. It comes from the ArduPilot test code:
    https://github.com/diydrones/ardupilot/blob/master/Tools/autotest/common.py
    """
    dlat = aLocation2.lat - aLocation1.lat
    dlong = aLocation2.lon - aLocation1.lon
    return (math.sqrt(dlat*dlat) * 1.113195e5, math.sqrt(dlong*dlong) * 1.113195e5)

def condition_yaw(heading, relative=False):
    global vehicle

    if relative:
        is_relative=1 #yaw relative to direction of travel
    else:
        is_relative=0 #yaw is an absolute angle
    # create the CONDITION_YAW command using command_long_encode()
    msg = vehicle.message_factory.command_long_encode(
        0, 0,    # target system, target component
        mavutil.mavlink.MAV_CMD_CONDITION_YAW, #command
        0, #confirmation
        heading,    # param 1, yaw in degrees
        0,          # param 2, yaw speed deg/s
        1,          # param 3, direction -1 ccw, 1 cw
        is_relative, # param 4, relative offset 1, absolute angle 0
        0, 0, 0)    # param 5 ~ 7 not used
    # send command to vehicle
    vehicle.send_mavlink(msg)


def goto_position_target_local_ned(north, east, down):
    global vehicle

    if down >= -1:
        print "Bad altitude parameter!!"
        return

    """
    Send SET_POSITION_TARGET_LOCAL_NED command to request the vehicle fly to a specified
    location in the North, East, Down frame.
    """
    msg = vehicle.message_factory.set_position_target_local_ned_encode(
        0,       # time_boot_ms (not used)
        0, 0,    # target system, target component
        mavutil.mavlink.MAV_FRAME_LOCAL_NED, # frame
        0b0000111111111000, # type_mask (only positions enabled)
        north, east, down,
        0, 0, 0, # x, y, z velocity in m/s  (not used)
        0, 0, 0, # x, y, z acceleration (not supported yet, ignored in GCS_Mavlink)
        0, 0)    # yaw, yaw_rate (not supported yet, ignored in GCS_Mavlink)
    # send command to vehicle
    vehicle.send_mavlink(msg)
    print "Going to ", north, east, down

def differential_NED(north, east, down):
    global vehicle

    N = vehicle.location.local_frame.north
    E = vehicle.location.local_frame.east
    D = vehicle.location.local_frame.down
    print vehicle.location.global_relative_frame.lat
    print "NED: ", N, E, D
    print north, east, down
```

```python

    goto_position_target_local_ned(N + north, E + east, D + down)

# Function to translate the UAV some amount forward, to the right, and down
def differential_FRD(front, right, down):
    global vehicle

    yaw = vehicle.attitude.yaw #radians
    E = right * math.cos(yaw) - front * math.sin(yaw)
    N = right * math.sin(yaw) + front * math.cos(yaw)

    differential_NED(N, E, down)

def center():
    global last_image_location, last_centering_time

    # Make sure this routine doesnt get called more than once every five seconds.
    if time.time() - last_centering_time < 5:
        return False
    else:
        last_centering_time = time.time()

    print "Centering..."

    alt = vehicle.location.global_relative_frame.alt
    FOV = 48.1 # Degrees
    angle_345 = 36.87 # degrees

    # Calculate the actual viewing X,Y distances
    X = 2 * alt * math.tan(math.radians(FOV/2)) * math.cos(math.radians(angle_345))
    Y = 2 * alt * math.tan(math.radians(FOV/2)) * math.sin(math.radians(angle_345))

    try:
        (cx, cy) = last_image_location.get_nowait()
        print cx, cy
    except:
        print "No image data."
        return False

    # Calculate the actual distance between the drone the the target
    # Scale the pixel location to the real location
    right = (cx - 320) * X / 640
    front = (-cy + 240) * Y / 480
    print "Center (FRD): ", (front, right)

    if (abs(front) <= 0.5) and (abs(right) <= 0.5):
        print "Centered!"
        return True
    else:
        differential_FRD(front, right, 0)
        return False

def shell_handler(command):
    global gps_coordinates, ignore_target, vehicle, shell_commands

    print "Command recieved: ", command

    if command == "takeoff":
        # Blocking
        takeoff(6)

    elif command == "land":
        # Non-blocking
        return_to_launch()

    elif command == "end":
        # Kills everything
        end_flight()

    elif command == "stop":
        # Non-blocking
        stop()

    elif command == "circle":
        # Non-blocking
        circle_POI()

    elif command == "goto":
```

```python
        # Non-blocking?

        # Make sure there is a location to go to
        location = []
        try:
            location = gps_coordinates.get_nowait()
        except:
            print "No available GPS coordinate"
            return

        print location
        go_to_coordinate(location[0], location[1], altitude=6, speed=1)

    elif command == "ignore":
        ignore_target = True

    elif command == "search":
        ignore_target = False

    elif command == "clearq":
        clearGPSQueue()

    elif command == "print":
        printData()

    elif command == "override":
        ignore_target = True
        vehicle.mode = VehicleMode("LOITER")
        time.sleep(1)

    elif command == "drop":
        drop()

    elif command.split()[0] == "pos":
        vehicle.mode = VehicleMode("GUIDED")
        try:
            front = float(command.split()[1])
            right = float(command.split()[2])
            down = float(command.split()[3])
            differential_FRD(front, right, down)
        except:
            print "Poorly formatted."

    elif command.split()[0] == "yaw":
        try:
            angle = float(command.split()[1])
            condition_yaw(angle)
        except:
            print "Poorly formatted."

    elif command == "center":
        while center() == False:
            if check_user_control() == True:
                print "User override break"
                break
            else:
                try:
                    data = shell_commands.get_nowait()
                    shell_commands.put(data)
                    print "Shell break"
                    break
                except:
                    pass
        print "Done centering."

    else:
        print "Not a vaild command."


def process_image_data():
    global image_data, identified, shutdown, image_socket, last_image_location

    while True:
        try:
            client_socket, address = image_socket.accept()
            print "Image socket connected from ", address
            break
        except:
            pass

    while True:

        # If it is time to shutdown
        with shutdown.get_lock():
            if shutdown.value == 1:
                print "Shutting down."
                break

        # image_socket is non-blocking, so an exception might be raised if there is no data in the socket
        try:
            # Get the data
            data_string = client_socket.recv(512)

            # Clear the current data in the shared queue
            try:
                image_data.get(block=False)
            except:
                pass

            # Add the new data
            data = pickle.loads(data_string)
            image_data.put(data)

            # If the target has been seen, set the flag
            with identified.get_lock():
                if data != -1:
                    print "Saw something!"
                    print data
                    try:
                        last_image_location.get_nowait()
                    except:
                        pass
                    last_image_location.put(data)
                    identified.value = 1
                else:
                    identified.value = 0
        except:
            pass



def process_gps_data():
    global gps_coordinates, shutdown, gps_socket

    while True:
        try:
            client_socket, address = gps_socket.accept()
            print "GPS socket connected from ", address
            break
        except:
            pass

    while True:
        # gps_socket is non-blocking, so an exception might be raised if there is no data in the socket

        # If it is time to shut down
        with shutdown.get_lock():
            if shutdown.value == 1:
                print "Shutting down"
                break

        try:
            data_string = client_socket.recv(512)
            data = pickle.loads(data_string)
            print data
            gps_coordinates.put(data)
        except:
            continue

def process_shell_data():
    global shell_commands, shutdown, shell_socket

    # Connect the shell socket
    # Must connect to shell before the UAV will arm
    while True:
        try:
```

```python
            client_socket, address = shell_socket.accept()
            print "Shell socket connected from ", address
            break
        except:
            pass

    while True:

        # If it is time to shut down
        with shutdown.get_lock():
            if shutdown.value == 1:
                print "Shutting down"
                break

        # shell_socket is non-blocking, so an exception might be raised if there is no data in the socket
        try:
            # Recieve data from shell socket
            data_string = client_socket.recv(512)
            data = pickle.loads(data_string)

            # Put it in the commands queue to be processed
            shell_commands.put(data)

        except:
            pass

def main():
    global image_socket, gps_socket, shell_socket, vehicle, identified, shutdown, ignore_target

    # Multi-core shared variables
    identified = Value('i', 0)
    shutdown = Value('i', 0)

    # Start the other scripts
    # os.system('python ../PiCamera/target_identification.py')
    # os.system('python ./shell.py')
    # os.system('python ../GoToHere/gotohere.py')

    # To open a terminal and run a command:
```

```
    # os.system("gnome-terminal -e 'bash -c \"sudo apt-get update; exec bash\"'")

    # Socket for listening to the target_identification script
    image_socket = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
    image_socket.setblocking(0)
    image_socket.bind(("",5001))
    image_socket.listen(5)

    # Socket for listening to the GPS coordinate script
    gps_socket = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
    gps_socket.setblocking(0)
    gps_socket.bind(("",5002))
    gps_socket.listen(5)

    # Socket for listening to the user shell
    shell_socket = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
    shell_socket.setblocking(0)
    shell_socket.bind(("",5003))
    shell_socket.listen(5)

    # Image information handler
    ImageProcess = Process(target=process_image_data)
    ImageProcess.start()

    # GPS information handler
    GPSProcess = Process(target=process_gps_data)
    GPSProcess.start()

    # Shell information handler
    ShellProcess = Process(target=process_shell_data)
    ShellProcess.start()

    # Initialize and arm the vehicle
    setup()

    # Time variable
    last_time = time.time()

    # Main control loop
```

```python
    while True:

        # See if there are any new commands queued up and act accordingly
        try:
            data = shell_commands.get_nowait()
            shell_handler(data)
        except:
            pass

        # Check if the circle needs to be expanded
        if SIM == False and vehicle.mode == "CIRCLE" and (time.time() - last_time > circle_period):
            update_circle_params()
            last_time = time.time()
            print "Expanding circle."

        # Check and see if the target has been found
        # Stop only if it has and you want to stop
        with identified.get_lock():
            if identified.value == 1 and ignore_target == False:
                print "Target Found!!"
                stop()
                ignore_target = True

        # If it is time to shut down
        with shutdown.get_lock():
            if shutdown.value == 1:
                break

    # Wait for the child processes to terminate
    GPSProcess.join()
    print "GPS process shut down"
    ImageProcess.join()
    print "Image process shut down"
    ShellProcess.join()
    print "Shell process shut down"

    # Shutdown the sockets
    image_socket.shutdown(socket.SHUT_RDWR)
    gps_socket.shutdown(socket.SHUT_RDWR)
    shell_socket.shutdown(socket.SHUT_RDWR)

    # Close the sockets
    image_socket.close()
    gps_socket.close()
    shell_socket.close()

    exit()


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

Listing B.6: Flight control interface code

B.6 Target Identification Code

```
#!/usr/bin/python
# import the necessary packages
import argparse
import cv2
from picamera import PiCamera
from picamera.array import PiRGBArray
from time import sleep
import numpy as np
import time
from multiprocessing import Queue, Process, Value
import sys
import Image
import StringIO
import time
import pickle
from BaseHTTPServer import BaseHTTPRequestHandler,HTTPServer
import socket
import signal

capture=None
client_socket = None
shutdown = None

# Inter-process queues
original_queue = Queue(maxsize=3)
final_queue = Queue(maxsize=3)

# Constants
RESOLUTION = 480

class CamHandler(BaseHTTPRequestHandler):
    def do_GET(self):
        if self.path.endswith('.mjpg'):
            self.send_response(200)
            self.send_header('Content-type','multipart/x-mixed-replace; boundary=--jpgboundary')
            self.end_headers()
            while True:
                try:

                    jpg = Image.fromarray(final_queue.get(block=True, timeout=None), 'RGB')

                    tmpFile = StringIO.StringIO()
                    jpg.save(tmpFile,'JPEG')
                    self.wfile.write("--jpgboundary")
                    self.send_header('Content-type','image/jpeg')
                    self.send_header('Content-length',str(tmpFile.len))
                    self.end_headers()
                    jpg.save(self.wfile,'JPEG')
                except KeyboardInterrupt:
                    break
            return
        if self.path.endswith('.html'):
            self.send_response(200)
            self.send_header('Content-type','text/html')
            self.end_headers()
            self.wfile.write('<html><head></head><body>')
            self.wfile.write('<img src="cam.mjpg"/>')
            self.wfile.write('</body></html>')
            return

def signal_handler(signal, frame):
    global shutdown

    with shutdown.get_lock():
        shutdown.value = 1

def identifySquare():
    global shutdown

    while True:

        with shutdown.get_lock():
            if shutdown.value == 1:
                break

        image = original_queue.get(block=True, timeout=None)

        # Cast the image to grayscale
        gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

        # Gaussian blur the image
        blur = cv2.GaussianBlur(gray, (7, 7), 0)

        # Detect the edges
        edge = cv2.Canny(blur, 50, 150)

        # find contours in the edge map
        (_, cnts, _) = cv2.findContours(edge, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)

        found_square = False

        # loop over the contours
        for c in cnts:
            # approximate the contour
            peri = cv2.arcLength(c, True)
            approx = cv2.approxPolyDP(c, 0.01 * peri, True)

            # ensure that the approximated contour is ”roughly” rectangular
            if len(approx) >= 4 and len(approx) <= 7:
                # compute the bounding box of the approximated contour and
                # use the bounding box to compute the aspect ratio
                (x, y, w, h) = cv2.boundingRect(approx)
                aspectRatio = w / float(h)

                # compute the solidity of the original contour
                area = cv2.contourArea(c)
                hullArea = cv2.contourArea(cv2.convexHull(c))
                solidity = area / float(hullArea)

                # compute whether or not the width and height, solidity, and
                # aspect ratio of the contour falls within appropriate bounds
                keepDims = w > 15 and h > 15
                keepSolidity = solidity > 0.8
                keepAspectRatio= aspectRatio >= 0.7 and aspectRatio <= 1.3

                # ensure that the contour passes all our tests
                if keepDims and keepSolidity and keepAspectRatio:
                    # draw an outline around the target and update the status
                    cv2.drawContours(image, [approx], -1, (255, 0, 0), 4)

                    try:
                        # compute the center of the contour region and draw the
                        # crosshairs
                        M = cv2.moments(approx)
                        (cX, cY) = (int(M["m10"] / M["m00"]), int(M["m01"] / M["m00"]))
                        (startX, endX) = (int(cX - (w * 0.15)), int(cX + (w * 0.15)))
                        (startY, endY) = (int(cY - (h * 0.15)), int(cY + (h * 0.15)))
                        cv2.line(image, (startX, cY), (endX, cY), (0, 0, 255), 3)
                        cv2.line(image, (cX, startY), (cX, endY), (0, 0, 255), 3)
                        center = (cX, cY)
                        # print center

                        # Serialize the data and stream it to the flight control code.
                        try:
                            data_string = pickle.dumps(center)
                            client_socket.send(data_string)
                            found_square = True
                        except:
                            pass

                    except Exception:
                        print "Divide by zero"

        final_queue.put(image, block=True, timeout=None)

        try:
            if found_square == False:
                data_string = pickle.dumps(-1)
                client_socket.send(data_string)
        except:
            pass

def putImage():
```

```python
    global RESOLUTION, shutdown

    # Set up the PiCamera
    camera = PiCamera()
    camera.framerate = 30

    if RESOLUTION == 1080:
        camera.resolution = (1920, 1080)
        rawCapture = PiRGBArray(camera, size=(1920, 1080))
    elif RESOLUTION == 720:
        camera.resolution = (1280, 720)
        rawCapture = PiRGBArray(camera, size=(1280, 720))
    elif RESOLUTION == 480:
        camera.resolution = (640, 480)
        rawCapture = PiRGBArray(camera, size=(640, 480))
    else:
        print 'Wrong Resolution'
        exit()

    # allow the camera to warmup
    sleep(0.1)

    # capture frames from the camera
    for frame in camera.capture_continuous(rawCapture, format="bgr", use_video_port=True):

        with shutdown.get_lock():
            if shutdown.value == 1:
                break

        frame = frame.array

        # Add the next image to process. If it blocks and times out, continue without adding a frame
        try:
            original_queue.put(frame, block=False)
        except:
            pass
        # clear the stream in preparation for the next frame
        rawCapture.truncate(0)

    # cleanup the camera and close any open windows
    camera.release()
    cv2.destroyAllWindows()


def displayImage():
    global shutdown
    last_time = 0
    while True:

        with shutdown.get_lock():
            if shutdown.value == 1:
                break

        # Get the final image to be displayed, if there is none, continue the loop
        final_image = final_queue.get(block=True, timeout=None)

        # show the frame and record if a key is pressed
        cv2.imshow("Frame", final_image)
        key = cv2.waitKey(1) & 0xFF # DONT DELETE NEED TO SHOW IMAGE

def main():
    global start_time, capture, img, client_socket, shutdown

    # Control-c handler to shut everything down properly
    signal.signal(signal.SIGINT, signal_handler)
    shutdown = Value('i', 0)

    # Start the client socket code to stream to the flight control script
    while True:
        try:
            client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
            client_socket.connect(('localhost', 5001))
            break
        except:
            print "Did not connect to sockets."

    # Start all the processes

    # Start the image identification processes
    P1 = Process(target=identifySquare)
    P2 = Process(target=identifySquare)
    P3 = Process(target=identifySquare)

    # Start the display process
    disp = Process(target=displayImage)

    # Start the raw image retrieval process
    capture = Process(target=putImage)

    P1.start()
    P2.start()
    P3.start()
    disp.start()
    capture.start()

    try:
        server = HTTPServer(('',8080),CamHandler)
        print "server started"
        server.serve_forever()
    except KeyboardInterrupt:
        capture.release()
        server.socket.close()

    P1.join()
    P2.join()
    P3.join()
    disp.join()
    capture.join()



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

Listing B.7: Target identification code

B.7 GPS Code

```python
import SimpleHTTPServer
import SocketServer
import cgi
import socket
import pickle

gps_sock = None

class ServerHandler(SimpleHTTPServer.SimpleHTTPRequestHandler):

    def do_GET(self):
        print("======= GET STARTED =======")
        print(self.headers)
        SimpleHTTPServer.SimpleHTTPRequestHandler.do_GET(self)

    def do_POST(self):
        global gps_sock
        print("======= POST STARTED =======")
        print(self.headers)
        form = cgi.FieldStorage(
            fp=self.rfile,
            headers=self.headers,
            environ={'REQUEST_METHOD':'POST',
                     'CONTENT_TYPE':self.headers['Content-Type'],
                     })
        print("======= POST VALUES =======")
        for item in form.list:
            print(item)
        latlong = float(form.getvalue('latitude')), float(form.getvalue('longitude'))
        data_string = pickle.dumps(latlong)
        print(data_string)
        gps_sock.send(data_string)


if __name__ == '__main__':
    global gps_sock
    PORT = 8081

    Handler = ServerHandler

    httpd = SocketServer.TCPServer(("", PORT), Handler)

    print "serving at port", PORT
    gps_sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    gps_sock.connect(('localhost', 5002))
    httpd.serve_forever()
```

Listing B.8: GPS input code

Appendix C

UAS Operation and Maintenance Procedure

Pre-Flight:

- Check charge on batteries. Anything over 4 volts per cell is ok to fly, though closer to 4.2 is preferred. Leave the battery monitor attached.
- Lock out all arms and make sure that the arm locks snap into place.
- Remove foam prop holders and fold out all props until they are straight.
- If props feel loose, tighten them. They should not be easy to twist.
- Power on transmitter and fix any errors that it displays onscreen.
- Plug in the battery to the rest of the S1000.
- PixHawk should beep a few times and after that the ESCs should beep roughly once a second.
- Hold down the blinking red button until the ESCs stop beeping. The UAS is now active but not armed.
- Have the pilot and spotter check the area for civilians, dogs or other things that may become a safety hazard or be harmed by the S1000.
- If area is clear, hold the throttle stick at zero throttle and push it all the way to the right. Hold there until the Pixhawk beeps. The pilot should inform people that the S1000 is armed. The UAS is now armed and ready to fly. It should automatically disarm after 5 seconds if the throttle is left at zero.

During Flight:

- Pilot should keep eyes on the UAS at all times.
- Spotter should keep an eye on the UAS as well.
- An extra person should run interference and handle social functions. This person should handle questions that are asked to the pilot and explain what is going on. If no third person is available, this job falls to the spotter.
- Once the battery alarm starts beeping, the UAS should be landed immediately. This should be around 3.6-3.4 volts, but the battery alarm can be set to go off at any point.
- Both pilot and spotter should check the landing area is clear before landing. Return To Launch can be used but in windy environments or odd payloads, a manual landing is most likely safer.

Landing and Pack Up:

- Pilot should land UAS and disarm, then let everyone know when it is safe to approach.
- Unplug everything from the battery.
- Turn off the transmitter after the UAS is turned off.
- Fold all props and put them back into the foam holders.
- Lower arms.

In case of crash:

- DO NOT APPROACH UNTIL PILOT HAS DISARMED UAS
- Once disarmed, approach until battery can be observed. If battery is swelling or punctured, keep a safe distance until you are sure that the battery is not releasing smoke or flame.
- If a battery discharges smoke, avoid breathing it at all costs! Lithium smoke destroys the bronchioles and causes permanent loss of lung capacity. This is more important for those on the swim team.
- If battery appears to be fine, a full check of UAS should be done. If any significant damage, stop flights. If any damage to props, replace them. Damage to props may be more significant than realized due to the stress they are under. Check damaged props later in a lab for usability considerations.
- Know who to contact in case the drone goes down in a tree or on top of a building, or any other unrecoverable location.

Appendix D

Flight Safety

This appendix discusses the flight safety procedures that were developed and followed as a part of this project.

D.1 Flight

D.1.1 Preparation

- Check batteries are charged before flying with the battery alarm
- Each cell should read at 4.2 volts max, 3.6-3.4 is low and 3.2-3.0 volts is considered dead
- Check the hand-held radio transmitter batteries are charged
- Check the airframe for loose screws, vibration dampeners that have fallen out, etc.
- Check props for damage from the last flight
- Ensure that you have the FAA registration number clearly printed on the aircraft, and that you have proof of registration with you: `https://www.faa.gov/uas/registration/`
- The pilot should be an AMA (Academy of Model Aeronautics) Member: `https://www.modelaircraft.org/joinnew.aspx`
- This safety code should be followed at all times as well: `https://www.modelaircraft.org/files/105.pdf`
- Check to ensure that there are no current flight restrictions in the area you plan to fly
- Ensure that you have the permission of the landowner whose land you are about to fly over, assuming you are not flying over public property

- If possible, use the FAA B4UFLY App as an additional resource to check that flight areas are clear and you are safe to fly

Ground Crew

- The pilot should always have their hands on the controller and always pay attention to the UAS and nothing else, this person should also be an AMA member
- The spotter keeps their eyes on the equipment and the UAS to warn the pilot of things the pilot may not see that could interfere with the flight
- The interference person should keep civilians/passersbys away from UAS flight zone and equipment.
- Add additional people as required, though having more than one person running interference is advised
- Pilot should relay commands to ground crew such as auto mode on/off, arm/disarm, etc. and if spotter is using equipment they should inform ground crew of what they are doing as well
- Simple radios such as a family radio system (FRS) are a good idea to have on hand

D.1.2 Takeoff

- The hand-held radio controller must always be turned on first
- All people or animals must clear a 5 meter radius from the UAS before arming can begin
- Rule of thumb: never get closer to UAS than the pilot or most knowledgeable person there
- Pilot alerts all ground crew that he/she is arming and taking off before doing so
- Random civilians wandering through and asking questions are ok so long as they dont interfere with anything or enter dangerous areas.
- Social functions and keeping an eye on civilians should fall to the Interference person(s) and no-one else. Interference person should stop people from talking to the pilot.

D.1.3 Flying Safety

- NO MATTER WHAT, no-one approaches the UAS until the pilot has CONFIRMED it is disarmed
- Follow FAA and AMA Guidelines, which are linked above

D.1.4 Breakdown

- Unplug everything from the battery, even the battery alarm as they draw power from only one cell and can unbalance and harm a battery if left in for too long
- Pack up all equipment and leave the site as it was found
- Log the flight in the logbook

D.1.5 Crash Landing

- DO NOT APPROACH UNTIL PILOT HAS DISARMED UAS
- Once disarmed, approach until the battery can be observed. If battery is swelling or punctured, keep a safe distance until you are sure that the battery is not releasing smoke or flame.
- If a battery discharges smoke, avoid breathing it at all costs! Lithium smoke destroys the bronchioles and causes permanent loss of lung capacity.
- If battery appears to be fine, a full check of the UAS should be done. If any there is any significant damage, stop flights. If there is damage to props, replace them. Damage to props may cause more problems than one might expect due to the stress props are put under. Check damaged props later in a lab for usability considerations.
- Know who to contact in case the drone goes down in a tree or on top of a building, or any other unrecoverable location.

D.2 Batteries

D.2.1 Storage

- Chargers have a Li-Po Storage option that should be used if the UAS if being left alone for more than a month or two such as summer break. Otherwise this is not necessary.

D.2.2 Charging

- Make sure the charger amp output is set to the same or lower than the battery capacity in amp-hours.
- Make sure the cell count on the charger is same as cell count on battery.

- Place the battery in metal bucket with some sand in bottom, ideally with a small covering on the sand to keep the sand from getting inside the battery casing
- Log which battery is charged in the battery logbook.

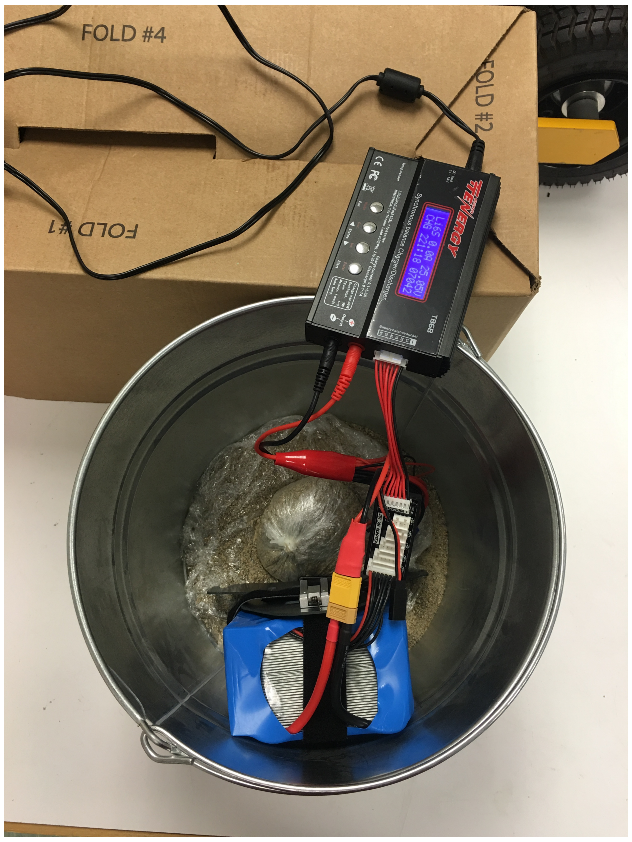


Figure D.1: Battery safely charging in charging bucket.

D.2.3 Safety

D.2.4 Physical signs of damage: Swelling

- If the battery feels squishy, although some batteries have a plastic cover that can bend outward, so be careful when checking for squishy-ness
- Punctures that reach the actual battery cells instead of just the casing
- Battery is hot, although in normal use a battery may become warm

D.2.5 Electrical signs of damage and electrical safety:

- Cell voltage imbalance - cells voltages are more than 0.2 V apart and the battery alarm will alert you to this
- Keep sand on hand as well as class D extinguisher if possible as bad chargers or damaged batteries in a charger can lead to lithium metal fires
- DO NOT TRY TO EXTINGUISH A LIPO BATTERY FIRE UNLESS YOU HAVE SAND OR A CLASS D EXTINGUISHER RATED FOR LITHIUM.
- LIPO BATTERIES RELEASE TOXIC GAS WHEN BURNING, DO NOT STAY NEAR THEM AND DO NOT BREATHE THE SMOKE
- Leave the building immediately and pull a fire alarm, try to alert authorities if possible about the situation
- Damaged batteries should be soaked for 24 hours in a super salinated solution, and then punctured by driving a nail through all cells. After this the battery can be disposed of normally.

D.3 Props

- Props should not have chunks taken out of them or small pieces of the plastic sticking out that could cause turbulence.
- Props should not have plastic deformation marks on them
- Small nicks in the props are ok but but not ideal, replace these props if possible
- Do not over-tighten props as this may result in the props shattering in flight
- Loose prop nuts will lead to the props decoupling in flight
- ESCs SHOULD NEVER BE POWERED INDOORS UNLESS PROPS ARE REMOVED
- If motors are being run without props, run them for as short a time as possible. The props provide cooling for the motors and without that cooling the motors will overheat and damage the motor and ESC.