



BRIGHAM YOUNG UNIVERSITY  
AUVSI CAPSTONE TEAM (TEAM 45)

---

## System Refinement Artifacts

---

## Table of Contents

---

### **PRIMARY ARTIFACTS**

---

Capstone Project Contract

Design Summary

Design Description

System Performance Summary

### **System Requirements Matrix**

---

### **SUPPORTING ARTIFACTS**

---

Refined Bill of Materials

Components Placement

Wiring Diagram

Airframe Model: Wing Extensions

Airframe Model: XFLR5

Airframe Tail Wedge Design, Construction, and Validation

Flight Test Log

Field Flight Checklist

Imaging Subsystem Description

Imaging Requirements Matrix

Imaging GUI API

Imaging GUI-Server Interface API

Geolocation Algorithm Description

Imaging Server API

Imaging Test Procedures



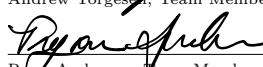
BRIGHAM YOUNG UNIVERSITY  
AUVSI CAPSTONE TEAM (TEAM 45)

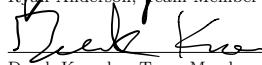
---

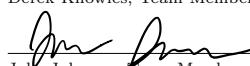
## Capstone Project Contract

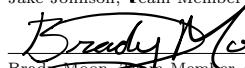
---

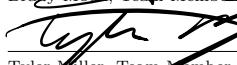
  
Andrew Torgeson, Team Member

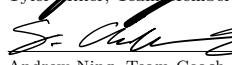
  
Ryan Anderson, Team Member

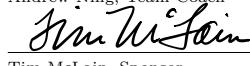
  
Derek Knowles, Team Member

  
Jake Johnson, Team Member

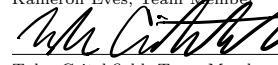
  
Brady Moon, Team Member

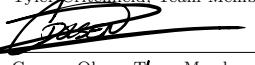
  
Tyler Miller, Team Member

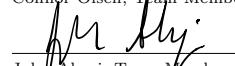
  
Andrew Ning, Team Coach

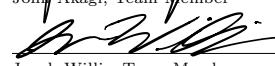
  
Tim McLain, Sponsor

  
Kameron Eves, Team Member

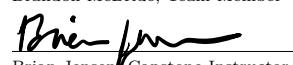
  
Tyler Critchfield, Team Member

  
Connor Olsen, Team Member

  
John Akagi, Team Member

  
Jacob Willis, Team Member

  
Brandon McBride, Team Member

  
Brian Jensen, Capstone Instructor

## Contents

Revision History . . . . .	2
Introduction . . . . .	2
Project Objective Statement . . . . .	3
Contact Information . . . . .	3
Project Approval Matrix . . . . .	4
Key Success Measures . . . . .	6
Change Management Procedure . . . . .	6

## Revision History

ID	Rev.	Date	Description	Author	Checked By
PC-444	1.0	10-02-2018	Opportunity development initial stage	Andrew Torgesen	Kameron Eves & Ryan Anderson & Jacob Willis & Tyler Critchfield & John Akagi
PC-444	1.1	10-17-2018	Added Key Success Measure explanations	Andrew Torgesen	Jacob Willis

## Introduction

Each year, the Association for Unmanned Vehicle Systems International (AUVSI) hosts a Student Unmanned Aerial Systems (SUAS) competition. While each year's competition has unique challenges, the general challenge is to build an Unmanned Aerial System (UAS) capable of autonomous flight, object detection, and payload delivery. This year's competition will be held June 12<sup>th</sup> to 15<sup>th</sup>, 2019 at the Naval Air Station in Patuxent River, Maryland.

The UASs entered into the competition are judged primarily on their mission success during the competition. Each team is also required to submit both a report and a flight readiness review presentation. The report should justify the UAS decision, explain design trade-offs, demonstrate the team's engineering process, and highlight the capabilities of the UAS. The flight readiness review presentation demonstrates that the UAS is capable of safely completing the competition. The overall score for a team is based on a combination of the points from the mission, report, and presentation.

For the last two years BYU has sponsored an AUVSI team to compete in the competition. The 2017 team was primarily volunteer based and placed 10<sup>th</sup> overall while the 2018 team was a Capstone team and placed 9<sup>th</sup> overall. This year's team is also a Capstone team consisting of BYU Mechanical, Electrical, and Computer Engineering students and looks to place as one of the top five teams.

## Project Objective Statement

Improve upon last year's BYU AUVSI unmanned aerial system (UAS) by improving path planning, obstacle avoidance, visual object detection, and payload delivery by April 1, 2019 with a budget of \$3,500 and 2,500 man hours.

## Contact Information

Team Member Name	Team Position	Contact Information
Andrew Ning	Coach	aning@byu.edu 801-422-1815
Andrew Torgesen	Team Lead	andrew.torgesen@gmail.com 661-210-5214
Brandon McBride	Controls/Payload Team	brandon.mcbride4@gmail.com 801-520-9165
Derek Knowles	Vision Team	knowles.derek@gmail.com 405-471-4285
John Akagi	Controls/Payload Team	akagi94@gmail.com 858-231-4416
Brady Moon	Controls/Payload Team	bradygmoon@gmail.com 435-828-5858
Tyler Miller	Vision Team	tylerm15@gmail.com 385-399-3472
Ryan Anderson	Controls/Payload and Airframe Team	rymanderson@gmail.com 208-789-4318
Jake Johnson	Vision Team	jacobcjohson13@gmail.com 801-664-7586
Tyler Critchfield	Controls/Payload and Airframe Team	trcritchfield@gmail.com 206-939-8274
Jacob Willis	Controls/Payload Team and Safety Officer	jbwillis272@gmail.com 208-206-1780
Connor Olsen	Vision Team	connorolsen72@gmail.com 385-230-3932
Kameron Eves	Controls/Payload Team	ccackam@gmail.com 702-686-2105

## Project Approval Matrix

The Project Approval Matrix, as depicted in Table 1, lists the major stages of development for the project, as well as their due dates and constituent artifacts. A budget is also included for each stage.

*Table 1: Project Approval Matrix for the UAS*

<b>Development Stage</b>	<b>Expected Completion Date</b>	<b>Design Artifacts Required for Approval</b>	<b>Budget</b>
Opportunity Development	October 5, 2018	Project Contract System Requirement Matrix Last Year Results Scoring Breakdown	\$100
Concept Development	November 2, 2018	Description of Vision Concept Description of Unmanned Ground Vehicle (UGV) Concept Description of Airframe Concept Test Procedures and Results Concept Selection Matrices Subsystem Interface Definitions	\$500
Subsystem Engineering	January 18, 2019	Wiring Diagram Vision Logic Diagram Autopilot Logic Diagram Bill of Materials UGV CAD Model UGV Drop Model Subsystem Requirement Matrices Subsystem Test Procedures and Results	\$2,000
System Refinement	March 22, 2019	Refined Integrated System Definition System Requirement Matrix UGV Engineering Drawings Refined Bill of Materials Integrated System Test Procedures and Results	\$800
Final Reporting	April 1, 2019	Final Report Compilation Flight Readiness Video Technical Design Paper Safety Pilot Log Team Promotional Video	\$100

## Key Success Measures

We developed a system requirements matrix in conjunction with the AUVSI competition rules (see artifact RM-001). All system-wide performance measures were considered, and five measures listed in Table 2 were selected as key success measures. Over the course of the next two semesters, we will gauge the desirability of our product based on how well the product completes each of these performance measures. Each performance measure will be evaluated in an environment designed to mimic the competition.

*Table 2: Key success measures for the UAS*

Measures (units)	Stretch Goal	Excel- lent (A)	Good (B)	Fair (C)	Lower Ac- cept- able	Ideal	Upper Ac- cept- able
Obstacles Hit (#)	0	1	3	5	0	0	5
Average Way- point Proxim- ity (ft)*	5	20	25	30	0	0	100
Characteris- tics Identified (%)**	80	40	30	20	20	100	100
Airdrop Ac- curacy (ft)	5	25	50	75	0	0	75
Number of Manual Takeovers	0	1	2	3	0	0	3

\* *Average Waypoint Proximity* refers to the norm of the distance between the UAS and the waypoint location at the point when the autopilot considers the waypoint to be captured.

\*\* *Characteristics Identified* refers to the ability to classify the color, shape, and textual content of visual targets scattered on the ground using camera measurements.

## Change Management Procedure

An Engineering Change Order (ECO) will be used to facilitate the proposal, approval, and implementation of any future changes to this contract. The ECO template is found on page

249 of the Product Development Reference (Mattson and Sorenson). A change is initiated by filling out the template and submitting it to all involved parties for approval. Upon unanimous approval, this contract will be edited, the version number will be changed, and the revision history section will be updated with the relevant information, including a reference to the ECO created.



BRIGHAM YOUNG UNIVERSITY  
AUVSI CAPSTONE TEAM (TEAM 45)

---

## Design Summary

---

ID	Rev.	Date	Description	Author	Checked By
DS-001	0.1	04-2-19	Design Summary Init	John Akagi	Brady Moon

## 1 Introduction

The BYU AUVSI Capstone team is competing in the AUVSI-SUAS 2019 competition this summer. The mission portion of the competition requires a small unmanned aircraft system (UAS) to autonomously fly to given waypoints, avoid imaginary obstacles, identify visual and geospatial characteristics of objects on the ground, and accurately drop a payload consisting of an unmanned ground vehicle (UGV) that is capable of autonomously driving to another location. Our team consists of four primary subteams: Airframe, Controls, Vision, and UGV. This document summarizes the description and performance of our system.

## 2 Design Description

In order to complete the AUVSI-SUAS objectives, we base our design around the Nimbus Pro airframe. This airframe has the internal capacity to carry all necessary instruments, batteries, and payloads needed for the competition. It also has the necessary aerodynamic features to fly for an extended period of time while maintaining stability in wind and moderately aggressive maneuvers. The aircraft is controlled using an autopilot based on Dr. McLain and Dr. Beard's *Small Unmanned Aircraft* book. The autopilot receives mission objectives from the judges and converts those to safe, flyable paths. While the aircraft is flying, pictures are taken and sent to a ground station where they are autonomously processed to identify shapes and letters on the ground. Finally, the autopilot creates a model of the current winds and uses that model to deliver a payload via parachute to a precise location.

## 3 Summary of Final Performance

We tested each of our designed components through flight tests. In total, we had almost three hours of manual flight and 0.33 hours of autonomous flight. We successfully planned and flew waypoints in both simulation and hardware. In simulation the airplane was able to fly waypoints within around a meter. In hardware, we were able to fly waypoints within 5 meters. The loss in accuracy is due to the addition of wind and other disturbances. We successfully planned and dropped our payload in simulation and hardware. In simulation we do not consider disturbances in wind or incorrect groundspeed values, so we simulate a payload drop with perfect accuracy. In hardware our accuracy was 17.7m, within the competition's 23m scoring boundary. The image classification system was also tested for its ability to detect and classify targets both manually and autonomously. The manual

system has proved to be effective for easy classification and submission of targets. The autonomous system is able to detect targets with an accuracy of about 90% and classify them with an accuracy of about 70%.

## 4 Conclusion and Recommendations

Results from both simulation and flight testing suggest that our integrated design is desirable (as outlined in artifact SP-002) and ready to perform in a mock-competition setting, pending a fix to a small piece of needed hardware. Through flight testing, the importance of thoroughly performing system testing and checks using the Field Flight Checklist (artifact PF-001) has been thoroughly learned. Our principle issues that have arisen in testing have been the occasional unreliability of our Ubiquiti WiFi and RC transmitter connections, as detailed in our Flight Log (artifact AF-004). To resolve the WiFi issue, we recommend obtaining and learning the available software debugging tools from Ubiquiti, called *AirOS*. These tools allow for WiFi site surveying and connection speed testing for both the rocket (antenna) and bullet (receiver) hardware components. With regards to the RC transmitter connectivity issue, we are committing to performing an RC interference analysis as part of our Field Flight Checklist before flying.

In the coming days, we will perform a mock competition in which all key success measures are tested during the same flight. Thus far, our flight tests have allowed us to obtain measured values for multiple key success measures at a time, though never all at once. The main barrier to a mock competition during the past week has been the aforementioned connectivity issues, which have only arisen recently and will be amended shortly.

Before the competition, the goal is to repeatedly test all subsystems together in a mock competition setting, using actual hardware testing data to iterate on our design and prove reliability over many more autonomous flights. If this is done, we are extremely confident that we will perform very well in the AUVSI-SUAS competition.



BRIGHAM YOUNG UNIVERSITY  
AUVSI CAPSTONE TEAM (TEAM 45)

---

## Design Description

---

ID	Rev.	Date	Description	Author	Checked By
DD-001	0.1	04-02-19	Initial release	Derek Knowles	Jacob Willis
DD-001	0.2	04-02-19	Added details	Jacob Willis	Derek Knowles

## 1 Introduction

A summary of each subteam's design description is provided in the sections below. The subsystems described make up the whole system that will be used at the AUVSI-SUAS 2019 competition.

## 2 Airframe

The airframe subteam selected the Nimbus Pro airframe. It is a fixed-wing plane with a large storage capacity and large wing span made of polystyrene (Fig. 1). It was selected during Concept Development for its long span and spacious fuselage. Multiple flight tests have been performed with R/C control and the autopilot. We have also successfully tested system integration with the UGV drop and vision subsystems during flight. This airframe was selected for its wing span of 1.95 m, total aircraft length of 1.29 m, and internal capacity of approximately 8000 cm<sup>3</sup>. Additionally, it was chosen because by using an off the shelf model we can quickly repair or replace components in the event of crashes or failures. Some small modifications were made to the plane in order to improve its performance, namely increasing the horizontal stabilizer incidence angle, adding a camera viewport, and creating a payload bay drop door.



Figure 1: Fully-constructed Nimbus Pro airframe before its first flight.

### 3 Controls

The purpose of the controls subsystem is to provide the UAS with the ability to autonomously fly waypoints and avoid obstacles. This subsystem works based on the principles described in Dr. McLain and Dr. Beard's *Small Unmanned Aircraft* book. Mission objectives are obtained from the judge's server. A flight path is determined from the objectives using an Rapidly-exploring Random Tree (RRT) method which randomly grows a tree until a path that connects all desired objective points is determined. Once a path has been determined, the plane attempts to minimize the distance between its planned path and current location through a series of proportional-integral-derivative (PID) controllers that adjust the airplane control surfaces.

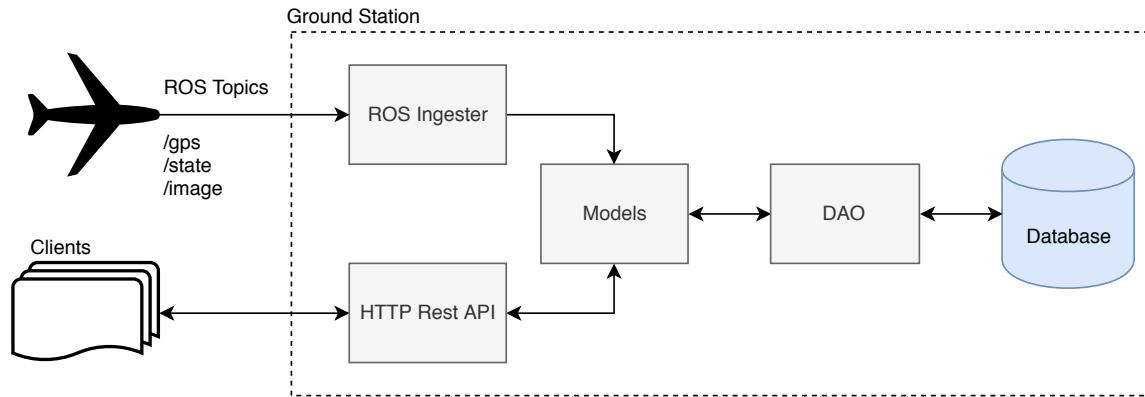
### 4 Vision

The main goal of the vision subsystem is to detect, classify, and geolocate ground targets on the competition field using a downward-pointing camera mounted to the airframe. This portion of the competition can be done either manually or autonomously, with more points being awarded for autonomously-detected targets. Our team chose to have a manual detection and classification system running in parallel with an autonomous system to maximize the amount of points awarded. A brief description of the different systems is given below.

In order for points to be awarded for autonomous submissions, there must be no human interference from the time the picture is taken to the time classifications are submitted to the judges. First, raw images are passed to a detector that finds regions of interest (ROIs). These ROIs are then examined to extract images of the target shape and letter, which are then classified by pretrained neural networks. The shape color and letter color are then identified and the classification is submitted to the judges.

The manual detection is performed in a similar manner, but with the identification and classification being done by a human client. This is performed using a GUI which allows up to three clients to work simultaneously, as well as monitor the autonomous submissions. Should the autonomous system register more than 10 false positives during the target detection sequence, a client will shut down the autonomous system, finishing up the target detection portion of the competition manually. In order to address concerns with the way the past year's team spread image classification across multiple machines making it difficult to identify bugs, replicate results and setup quickly, our team created a basic server-client architecture as defined as shown in Figure 2. The basic data flow of all these components is shown in Figure 3. Data flows back and forth between client and server,

with the server holding a definitive-final copy of a target image, as well as a history of its state during intermediate steps.



*Figure 2: Server Architecture*

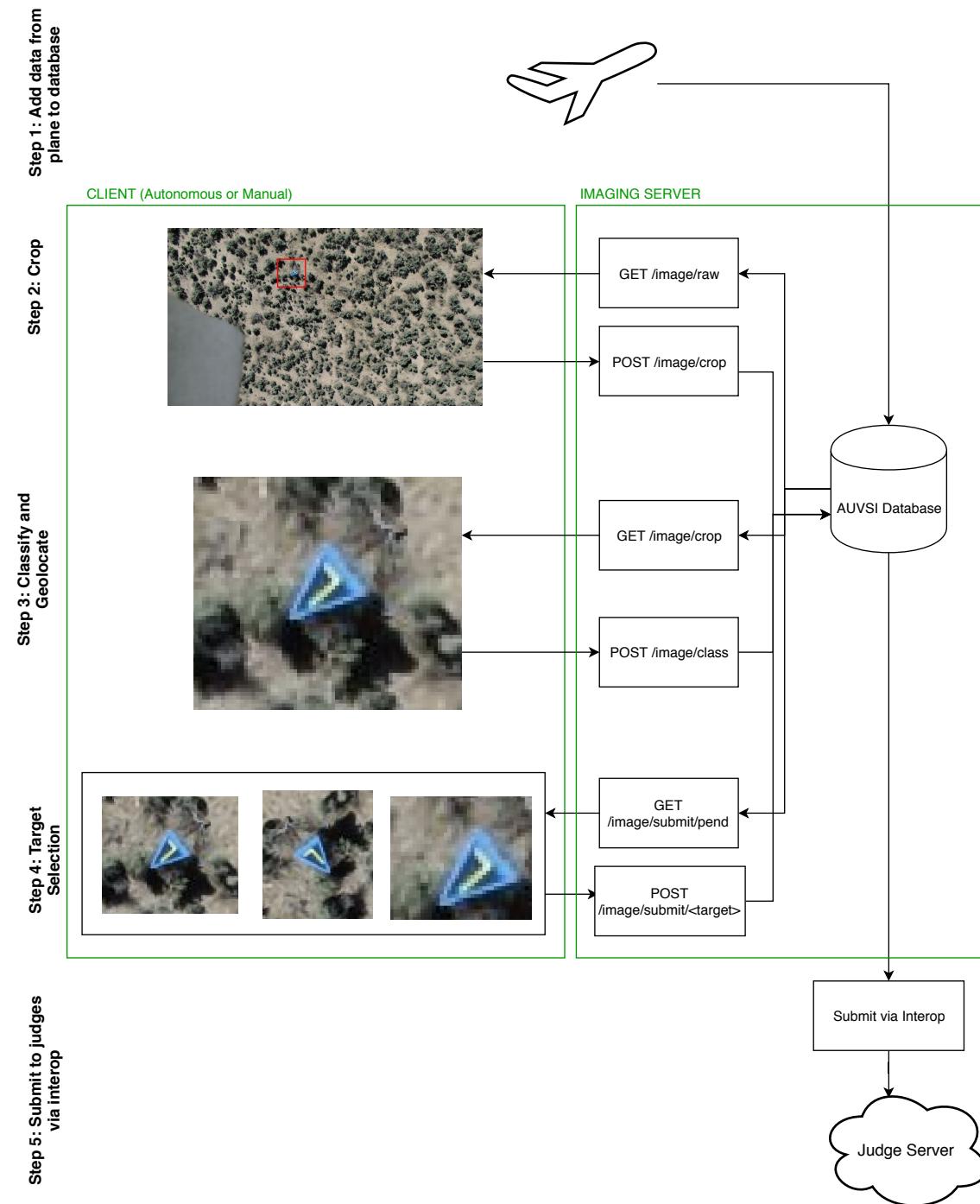


Figure 3: AUVSI Imaging Data Flow Through Autonomous/Manual Classification

## 5 UGV

The UGV is a ground vehicle that must be loaded within the aircraft prior to flight, and fly with the aircraft through the waypoint portion of the mission. Upon a command from the flight controller system, a small hatch opens and the UGV falls out. The UGV is carried to the ground by a lightweight 36 inch nylon parachute, purchased from FruityChutes. The parachute is loaded onto the aircraft in a tube that allows the UGV to pull it out of the aircraft as it falls. This helps stop the tangling that can come from a folded parachute. To also prevent tangling, and to make for a more predictable drop, the parachute is folding according to GV-007. After exiting the aircraft the parachute will be opened by drag. This will slow down the UGV enough to allow the it to survive impact without damage. A visual depiction of our chosen system can be seen in Fig. 4.



*Figure 4: Our parachute and simulated UGV as seen from the side.*

## 6 Conclusion

This year's AUVSI team has increased the desirability and transferability of the Unmanned Aircraft System. Each subteam has made progress on each of the key success measures thereby increasing the reliability of each subsystem.



BRIGHAM YOUNG UNIVERSITY  
AUVSI CAPSTONE TEAM (TEAM 45)

---

## Summary of System Performance

---

ID	Rev.	Date	Description	Author	Checked By
SP-001	0.1	04-02-19	Initial Commit	Brady Moon	Andrew Torgesen

## 1 Introduction

This artifact highlights of our testing and results for our integrated UAS. Through a combination of simulation and hardware flight testing, we are convinced that our integrated system, together with further refinements over the next two months to maximize our point potential, will obtain a favorable market response in the AUVSI-SUAS competition. Our testing procedures and results can be distilled into four main categories: Airframe, Controls, Vision, and UGV. Summaries of these four subsystem results are detailed below.

## 2 Summary of System Performance

### 2.1 Airframe

At the end of Fall semester, our new airframe suffered a devastating crash that left it in pieces. As a result, we confirmed the possibility of a quick two-day rebuild, and the airframe has since completed more than 20 flight tests (see Flight Log: artifact AF-004). It has successfully integrated with the imaging, autopilot, and UGV subsystems. It is light, weighing only 4.5 kg, and is spacious enough to contain all required components and more. It flies smoothly with plenty of thrust to safely take off with an endurance of approximately 50 minutes. It maintains stability despite active wind gusts of up to around 7 m/s. It flies slowly with an average flight speed around 14 m/s, resulting in crisp images and easy maneuvering. See artifact AF-001 for more detailed performance metrics.

### 2.2 Controls

#### 2.2.1 Stable Flight

Thus far, over the course of numerous flight tests, we have accrued 23 minutes of autonomous flight time in hardware. Most of this autonomous hardware flight time has been for the process of tuning our autopilot gains, as detailed in artifact CT-001. The process of tuning autopilot gains has been crucial to the success of our integrated system, as it ensures stable autonomous flight. Gain tuning has resulted in excellent stability performance in the absence of wind, as well as satisfactory performance in the presence of wind.

### 2.2.2 Waypoint Accuracy

In simulation, we were able to fly waypoints within approximately 1 meter. The results were obtained through a series of 5 trials. In each trial, 5 waypoints were randomly generated and the plane was instructed to find a path that connected all 5 points. Once the path was planned, the plane flew the path in simulation. The results of the five trials are shown in Table 1. For hardware, a similar approach was taken. Three points were located in the flight space which would be safe to fly through. Once the points were determined, the plane was commanded to plan a path through them and then fly. Due to disturbances such as wind, the plane was unable to fly the waypoints as accurately as in simulation but each waypoint was able to be flown within 5 meters. The waypoints, planned path, and flown path are shown in Figure 1 in yellow, green, and white, respectively.

*Table 1: Table of results showing the minimum distance between the UAV and each waypoint.*

Test No.	Waypoint 1	Waypoint 2	Waypoint 3	Waypoint 4	Waypoint 5
1	.750 m	.128 m	.092 m	.430 m	.688 m
2	1.058 m	.127 m	.180 m	.376 m	.328m
3	.602 m	.625 m	.158 m	.204 m	.122 m
4	.427 m	.351 m	.042 m	.018 m	69.7m
5	.152 m	.085 m	.011 m	1.11 m	.90 m

## 2.3 Vision

Through flight tests and post-flight analysis, we were able to autonomously detect 70% of ground targets with only 1-3 false positives. Manual classification was also able to classify 100% of targets it received during flight. The main performance bottleneck is image transfer speed from the plane to the ground. The camera is able to capture images at a high rate, but unable to stream them all back to the groundstation fast enough, causing us to loose information through flight.

For autonomous classification we were able to reliably classify 10/13 shapes and 18/26 letters. These statistics should improve as we refine our dataset and neural net models. With finetuning the day of flight, we are able to classify 7/10 colors.

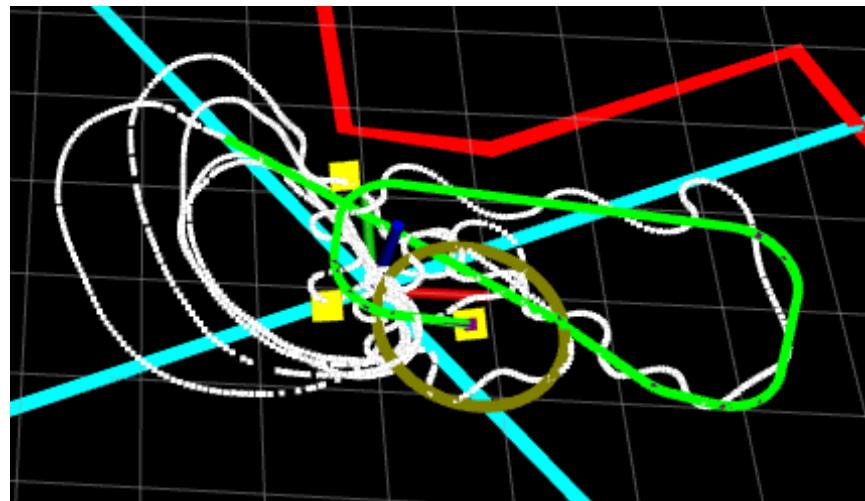
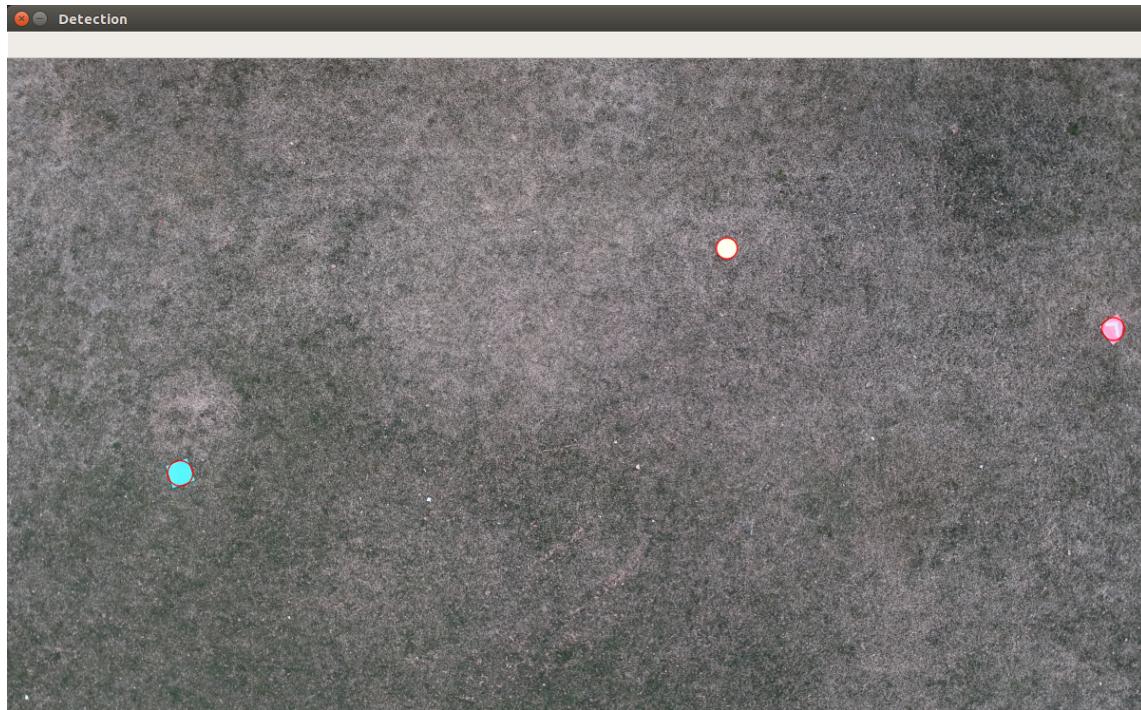


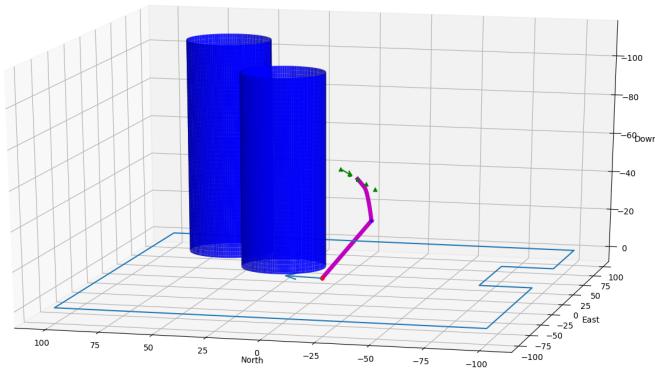
Figure 1: The flight path is shown for the hardware test. The yellow points mark the desired waypoints, the green line shows the desired path, and the white line shows the actual flown path. The golden circle shows the final loiter location for the plane to fly once the planned path is flown. As a result of strong winds, the plane had difficulty staying perfectly on the planned path but was still able to follow the path and hit the waypoints.



*Figure 2: Autonomous system detecting ground targets (red circles)*

## 2.4 UGV

Using our new simulation and payload drop planning algorithms, we dropped our payload twice with an average prediction accuracy of 17.7m. These drops were within our predicted performance values even though it was a gusty day. In simulation, our payload drop planning algorithms have consistently been able to plot paths that can drop the UGV while avoiding obstacles and competition boundaries.



*Figure 3: Payload Drop Plan Plot*

## 3 Conclusion

Testing for obtaining key success measure values has yielded favorable results in all categories. Through hardware flight testing, we have been able to test multiple subsystems during the same flight (such as Airframe-Controls-Payload and Airframe-Vision). However, we still need to perform a mock competition for testing all key success measures in one flight during a competition setting. This will be performed in the coming days.



BRIGHAM YOUNG UNIVERSITY  
AUVSI CAPSTONE TEAM (TEAM 45)

---

## UAS Requirements Matrix

---

<b>ID</b>	<b>Rev.</b>	<b>Date</b>	<b>Description</b>	<b>Author</b>	<b>Checked By</b>
RM-001	0.1	09-07-2018	Fall camp draft	Brady Moon	Jacob Willis
RM-001	0.2	09-14-2018	Revisions after design review	Derek Knowles	Kameron Eves
RM-001	1.0	10-08-2018	Expansion for stage approval	Kameron Eves	Brandon McBride
RM-001	1.1	10-08-2018	Reordered requirements to match priority	Jacob Willis	Brady Moon
RM-001	1.2	10-17-2018	Fixed inconsistency in autonomous flight requirement	Andrew Torgesen	Kameron Eves
RM-001	1.3	11-08-2018	Added Target Values	Kameron Eves	Andrew Torgesen
RM-001	2.0	04-05-2019	Added Measured Values	Brady Moon	Andrew Torgesen

## UAS Requirements Matrix

Product: UAS  
Subsystem: N/A

**Notes:**  
UAS = Unmanned Aerial System  
UGV = Unmanned Ground Vehicle

Measured	Predicted	Target Values	Upper Acceptable	Ideal	Lower Acceptable	Market Requirements		Performance Measures		Units
						Importance	Impact	Importance	Impact	
30	30	20	40	20	15	10%	●	1	Flight Time	Minutes
0	5	0	10	0	0	10%	●	2	Post Processing Time	Minutes
15	30	20	40	20	15	20%	●	3	Autonomous Flight Time	Minutes
100	100	100	100	100	100	20%	●	4	Percent of Waypoints Hit	Percent
4.57	3	0	5	0	0	20%	●	5	Average Minimum Distance to Waypoint	Meters
2	10	0	20	0	0	10%	●	6	Percent of Obstacles Hit	Percent
50	80	90	100	100	75	10%	●	7	Percent of Correct Characteristics Identified	Percent
0	80	100	100	100	75	20%	●	8	Percent of Images Correctly Geolocated	Percent
0	89	100	100	100	0	20%	●	9	Percent of Objects Submitted in Flight	Percent
90	80	100	100	100	75	40%	●	10	Percent of Objects Autonomously Submitted	Percent
58	25	5	75	5	0	40%	●	11	Payload Drop Distance to Target Location	Feet
Y	Y	Y	Y	Y	Y	60%	●	12	Complies with AMA Safety Code	Yes/No
3	2	0	15	0	0	60%	●	13	Penalties	Count
						Importance	Impact	Impact	Impact	Market Response
						6%	2%	8%	2%	Very Good
						10%	2%	10%	2%	Very Good
						10%	2%	10%	2%	Very Good
						10%	2%	10%	2%	Very Good
						10%	2%	10%	2%	Good
						10%	2%	10%	2%	Good

Figure 1: Top-level requirements matrix for the unmanned aerial system. Performance measures that are marked in grey are our key success measures. Note that cumulatively, our key success measures account for 46% of the possible points.



BRIGHAM YOUNG UNIVERSITY  
AUVSI CAPSTONE TEAM (TEAM 45)

---

## Bill of Materials

---

ID	Rev.	Date	Description	Author	Checked By
BM-001	0.1	02-19-2019	SS Engineering Draft	Tyler Critchfield	Brandon McBride
BM-001	1.0	04-02-2019	Complete System Bill of Materials	Jacob Willis	Jake Johnson

# 1 Introduction

The following tables show our Bill of Materials for the entire system. We've included the following information with each component: item number, item name, items interfaced with, quantity, mass, and vendor.

*Table 1: A bill of materials for the final flight system. Information listed includes the item number, name, quantity, mass, vendor, and a list of other items each interfaces with. Masses are not included if they were already included in another item's mass or if it is irrelevant (if the item is not in the plane). Not included is each component of the aircraft that came with the kit (e.g., control horns, glue, connector pieces, coverings, etc.).*

Item #	Item Name	Interfaces With	Quantity [#]	Mass [g]	Vendor
1	My Fly Dream Nimbus Pro Airframe	Everything	1	2000	BangGood
2	Pitot Tube	Item 3	1	19	Amazon
3	PX4 Airspeed Sensor	Item 4	1	N/A	N/A
4	Flip32 Breakout I2C	Items 5, 6, 7, 19	1	60	Custom
5	OpenPilot Revo Flight Controller	Item 4	1	35	Hyperion World
6	Servos	Item 4	4	44	Hobby King
7	ESC	Items 4, 8, 19	2	160	Hobby King
8	Motors	Item 7	2	280	Tower Hobbies
9	RC Receiver	Items 10, 19	1	19	Team Blacksheep
10	RC Antennas	Item 9	2	0	N/A
11	POE Adapter	Items 12, 15, 19	1	30	N/A
12	Ubiquiti Bullet	Items 11, 13	1	165	Amazon
13	5GHz antenna	Item 12	1	32	N/A
14	5GHz Litebeam Antenna	Items 1, 30	1	N/A	Newegg
15	Odroid XU4	Items 4, 11, 16, 17, 19	1	99	Hard Kernel
16	Sony A6000 Camera	Items 15, 28	1	241	Sony
17	InertialSense	Item 15, 18	1	N/A	N/A
18	GPS Antenna	Item 17	1	94	Drotek
19	Power Harness	Items 4, 7, 9, 11, 15, 20-22, 31	1	130	N/A

---

*Table 2: A continuation of the system bill of materials from the previous page.*

Item #	Item Name	Interfaces With	Quantity [#]	Mass [g]	Vendor
20	Castle BEC 20 A	Items 15, 19, 22, 39	1	N/A	Castle Creations
21	Castle BEC 10 A	Items 9, 19	1	N/A	Castle Creations
22	Bottle Drop Arduino	Items 15, 19, 23	1	N/A	Hobby King
23	Payload Release	Item 22	1	18	Hobby King
24	Tail Foam Wedge	Item 1	1	10	Custom
25	Battery Straps	Item 31	2	5	N/A
26	Velcro Strips	Items 5, 12, 31	6	N/A	EE Shop
27	Controller Board Housing		1	N/A	N/A
28	Camera Housing	Item 16	1	N/A	Custom
29	Propellers	Item 8	2	14	Hobby King
30	WiFi Router	Item 14	1	N/A	Amazon
31	4S Lipo Battery	Item 19, 25	1	944	Hobby King
32	UGV Chassis	Item 33, 37	1	88	Amazon
33	UGV Battery	Item 32	1	31	Amazon
34	UGV GPS	Item 32, 37	1	24	Amazon
35	Parachute housing	Item 36	1	10	Custom
36	36" Parachute	Items 32, 35	1	92	FruityChutes
37	UGV Controller	Items 32, 33, 34	1	35	Hyperion
38	Fiber Tape	Item 1	N/A	N/A	Amazon
39	Camera Power Supply	Items 16, 20	1	40	Amazon/Custom
40	UGV Payload Bay	Items 1, 23, 32, 36	1	N/A	Custom

[./Airframe/ComponentsPlacement/ComponentsPlacement.pdf](#)



BRIGHAM YOUNG UNIVERSITY  
AUVSI CAPSTONE TEAM (TEAM 45)

---

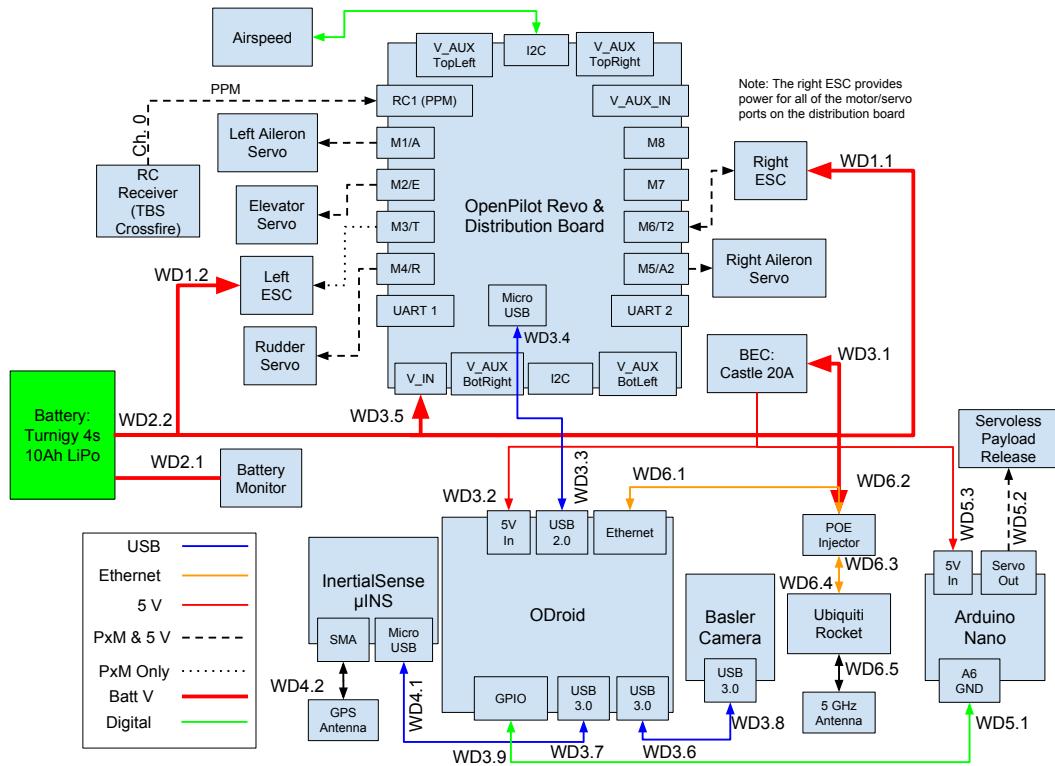
## Wiring Diagram

---

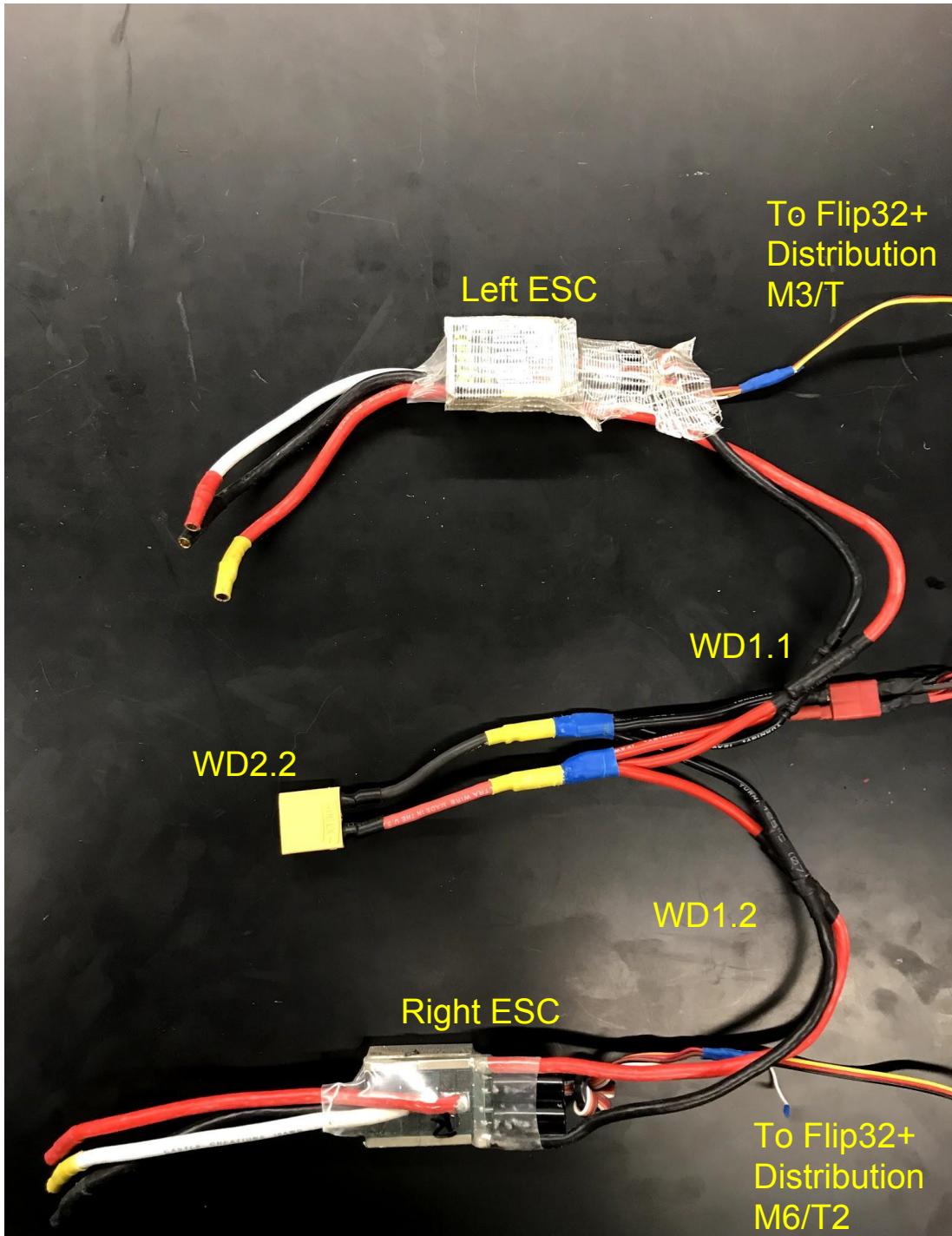
ID	Rev.	Date	Description	Author	Checked By
AF-005	0.1	12-10-2018	Created	Jacob Willis	Andrew Torgesen

## Introduction

This artifact depicts the wiring connections between electrical components within the airframe. Each connection label corresponds to a page in the diagram and a connection on that page, so WD5.1 refers to connection 1 on page 5. Connections that appear on more than one page are given the page number of the page that best depicts that connection. Page 2 is a logical diagram showing how all of the components are connected together. The remaining pages are labeled photographs of the connections.



*Top level logical diagram. The power harness consists of the Batt V connections, and the BEC 20A converter.*



WD1

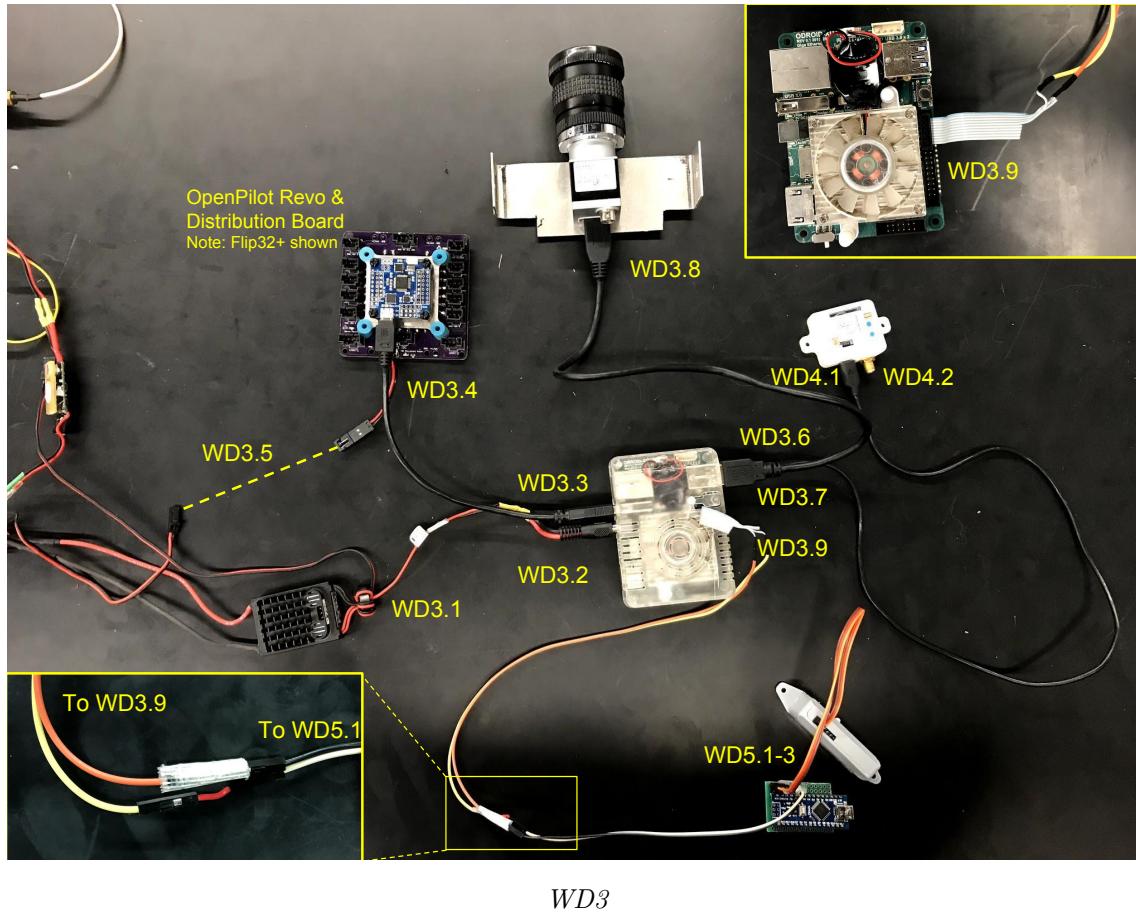
Wiring Diagram

---



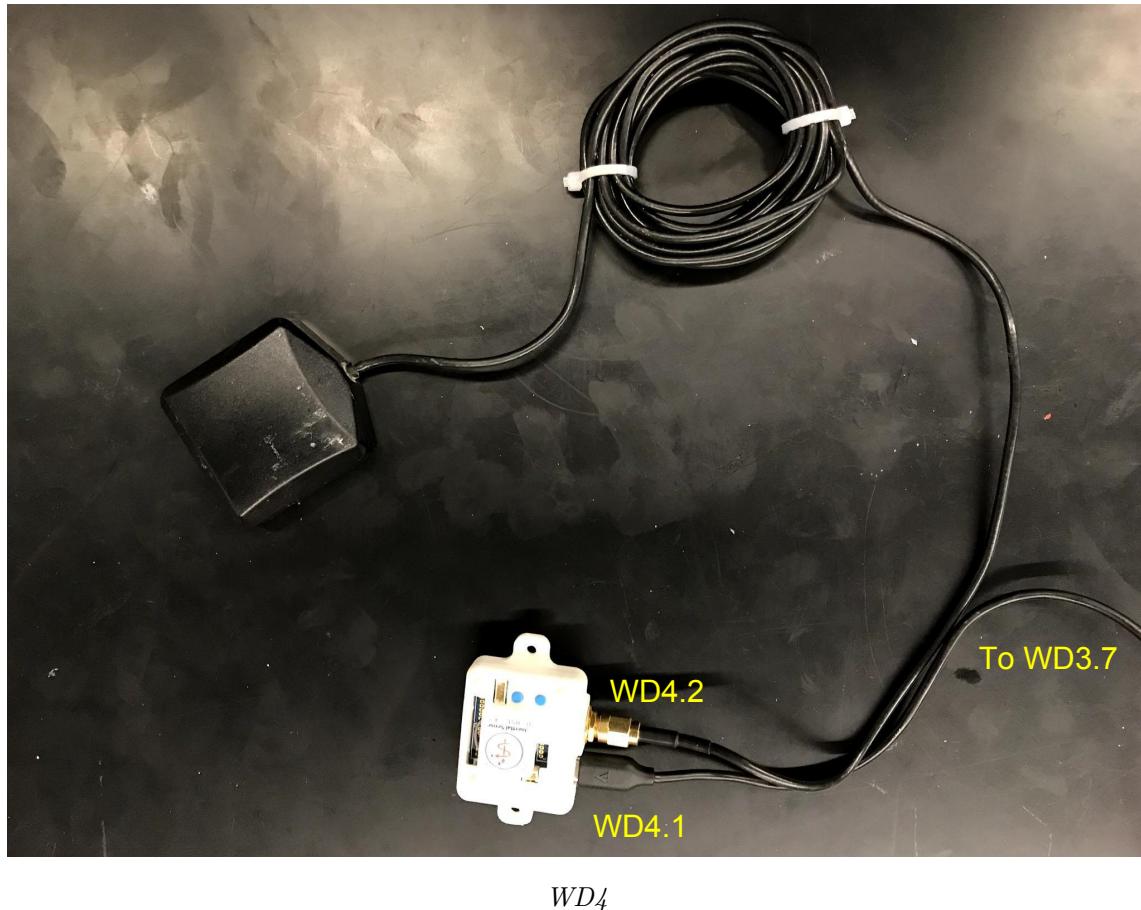
WD2

## Wiring Diagram



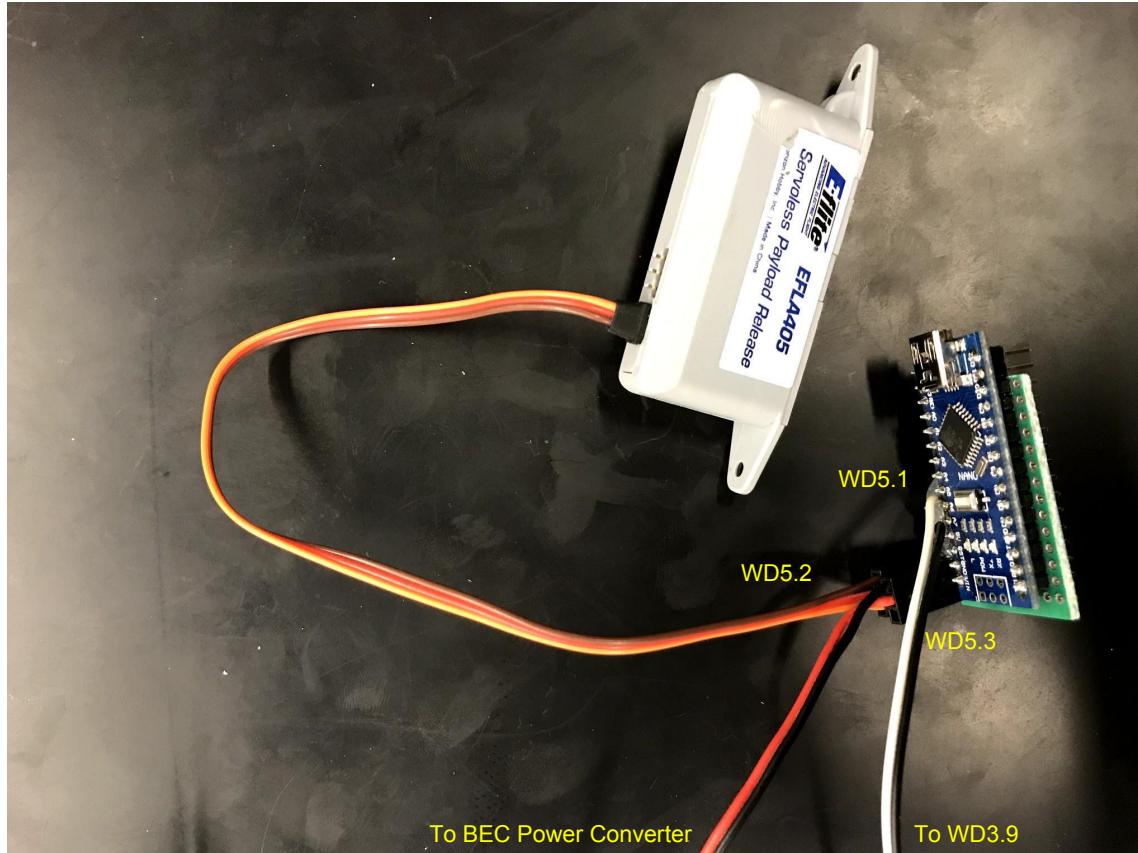
Wiring Diagram

---



Wiring Diagram

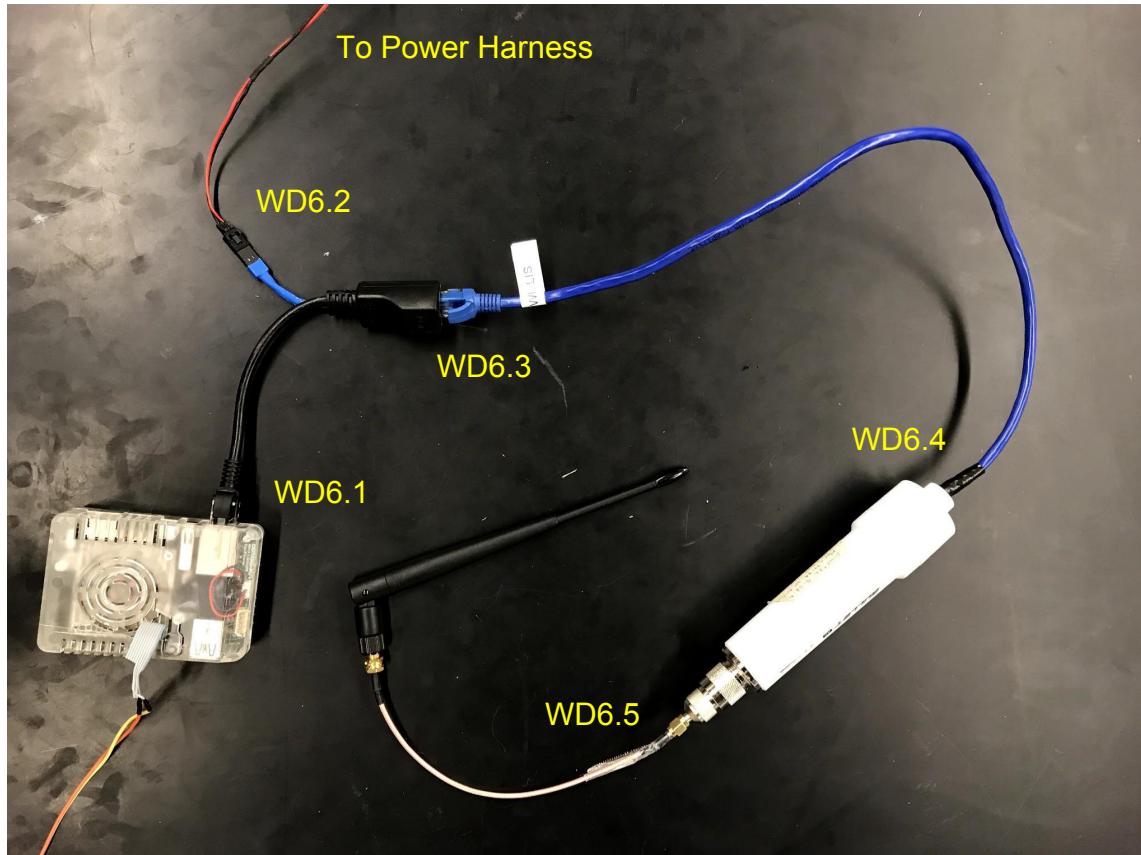
---



WD5

Wiring Diagram

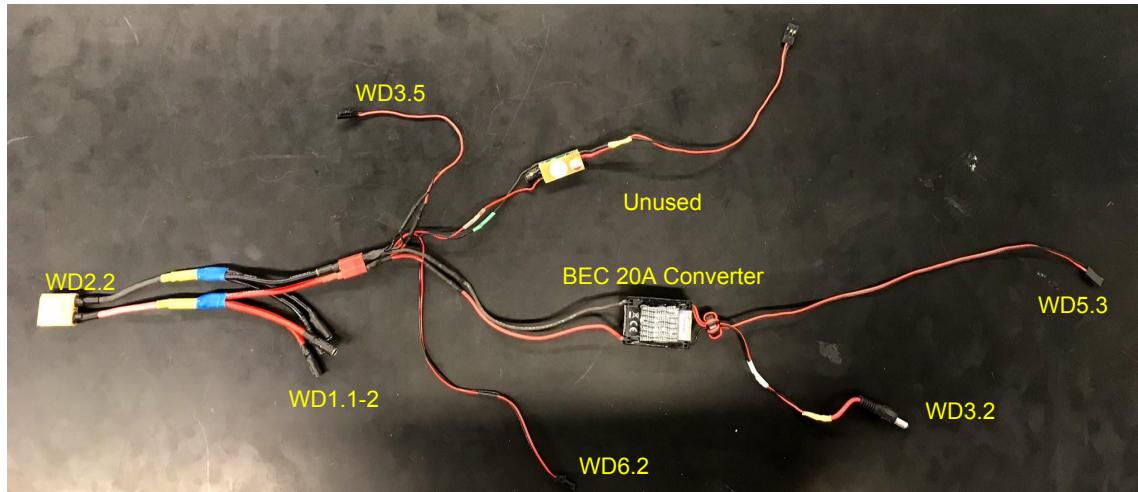
---



WD6

Wiring Diagram

---



*Power harness*



BRIGHAM YOUNG UNIVERSITY  
AUFSI CAPSTONE TEAM (TEAM 45)

---

## Wing Extensions Design Decision

---

ID	Rev.	Date	Description	Author	Checked By
AF-012	0.1	02-20-2019	Initial draft	Ryan Anderson	Tyler Critchfield

## 1 Introduction

In order to enhance the performance of our airframe, the XFLR5 model described in artifact AF-011 was extended to test the viability of wing extensions to further decrease the design speed of the airframe. As discussed in AF-011, a slower design speed will facilitate payload drop, waypoint navigation, and image capture- three of our key success measures. After this study, it was determined that wing extensions did not provide sufficient improvement to aerodynamic performance to warrant implementation.

## 2 Method

The decision not to implement wing extensions was reached by careful consideration of manufacturability and aerodynamic benefits, as follows.

### 2.1 Manufacturability

First, methods of adding wing extensions to the airframe were brainstormed. Each wing attaches to the fuselage by fitting two circular spars into two larger spars in the fuselage. A spring-loaded clipping mechanism is fitted to the root of each wing, and clips into the fuselage (see Fig. 1). It was determined that up to 5 cm of foam section could be cut out of foam and fitted onto the spars on the root side of the wings. The clipping mechanism would be removed and replaced on the outside of the added foam section. Any more than 5 cm of extensions would require extensions on the spars.



Figure 1: Clipping mechanism and spars for mounting wings. Fuselage is shown in this image.

## 2.2 Analysis

In order to determine the aerodynamic advantage of wing extensions, the wings in the XFLR5 model were lengthened and an analysis similar to that described in AF-011 performed to predict the resulting design velocity.

## 3 Results

The model results are tabulated in Table 1. Note that only a very small decrease in speed is awarded by significant extensions of the wings. Note also that the design speed without any wing extensions is already significantly slower than last year's design. As a result, we decided not to add wing extensions.

*Table 1: Effect of wing extensions on design speed.*

Extension per Wing (cm)	Design Speed (m/s)
0	12.863
5.0	12.675
7.5	12.565
10.0	12.459

## 4 Conclusion

In summary, the aerodynamic performance of the aircraft was insufficiently enhanced by wing extensions to make them worth our while. Since making the decision to not add wing extensions, the airframe has flown well and integrates reliably with the three other subteams, thus validating our design decision.



BRIGHAM YOUNG UNIVERSITY  
AUVSI CAPSTONE TEAM (TEAM 45)

---

## Airframe Models

---

ID	Rev.	Date	Description	Author	Checked By
AF-011	0.1	02-07-2019	Initial draft	Ryan Anderson	Tyler Critchfield

## 1 Introduction

In order to predict the performance of our selected airframe, we created an aerodynamics model of the airframe using the open-source package XFLR5. This informed design decisions including placement of the center of gravity (CG) and modifications to the tail in order to optimize performance. A summary of the design decisions made as a result of the XFLR5 model is included in the conclusion of this artifact for quick reference.

## 2 Objective

The objective of the XFLR5 model is to provide design guidance for the achievement of our key success measures, three of which rely significantly on the aerodynamic performance of the airframe. These include the following:

- Close waypoint proximity
- High percentage of characteristics identified
- High airdrop accuracy

Each of these characteristics will be benefited by a slower stall speed than last year. Waypoint proximity and characteristics identified can be further benefited by improving static and dynamic stability, as a stable plane is easier to control and provides a more stable imaging platform for crisper images. As a system-wide requirement, the aircraft must have sufficient endurance to accomplish the above-listed objectives. With this in mind, major aerodynamic design objectives for the airframe include a slower stall speed than last year (ideally on the order of 10-15 m/s), sufficient static and dynamic stability, and a high lift-to-drag ratio to improve endurance (see Airframe Requirements Matrix). The XFLR5 model allowed us to make design decisions such as tail incidence angle and CG placement to meet these requirements.

## 3 Method

The Nimbus Pro airframe was measured, analyzed, and modeled as follows:

### 3.1 Geometry

In order to capture the aerodynamic performance of the aircraft, the wing and tail were modeled, with an extra drag term added for the fuselage. The wing, elevator, and fin were measured and modeled in the Plane Editor with geometries defined in Tables 1, 2, and 3. Airfoils were approximated by NACA foils with corresponding max camber, max camber location, and max thickness relative to the chord. These were assumed to be constant

throughout the wings. Figure 2 shows a cross section of the wing used to determine its airfoil.

*Table 1: XFLR5 Wing Parameters*

Parameter (units)	Value
<b>y(m)</b>	[0.000 0.925 0.975]
<b>chord(m)</b>	[0.310 0.250 0.250]
<b>offset(m)</b>	[0.000 0.000 0.000]
<b>dihedral(°)</b>	[0.000 0.000 0.000]
<b>twist(°)</b>	[0.000 0.000 5.000]
<b>foil</b>	NACA 3311

*Table 2: XFLR5 Elevator Parameters*

Parameter (units)	Value
<b>y(m)</b>	[0.000 0.303]
<b>chord(m)</b>	[0.180 0.130]
<b>offset(m)</b>	[0.000 0.025]
<b>dihedral(°)</b>	[0.000 0.000]
<b>twist(°)</b>	[0.000 0.000]
<b>foil</b>	NACA 0010

*Table 3: XFLR5 Fin Parameters*

Parameter (units)	Value
<b>y(m)</b>	[0.000 0.270]
<b>chord(m)</b>	[0.220 0.120]
<b>offset(m)</b>	[0.000 0.110]
<b>dihedral(°)</b>	[0.000 0.000]
<b>twist(°)</b>	[0.000 0.000]
<b>foil</b>	NACA 0010

Also of note, the x location of the elevator and fin was measured to be at 0.690 m behind the wing. In order to prevent any possibility of computational singularities in the VLM solution, the elevator and fin were both placed 30 cm below the wing (i.e., z location was set to -0.300 m) as shown in Fig. 1.

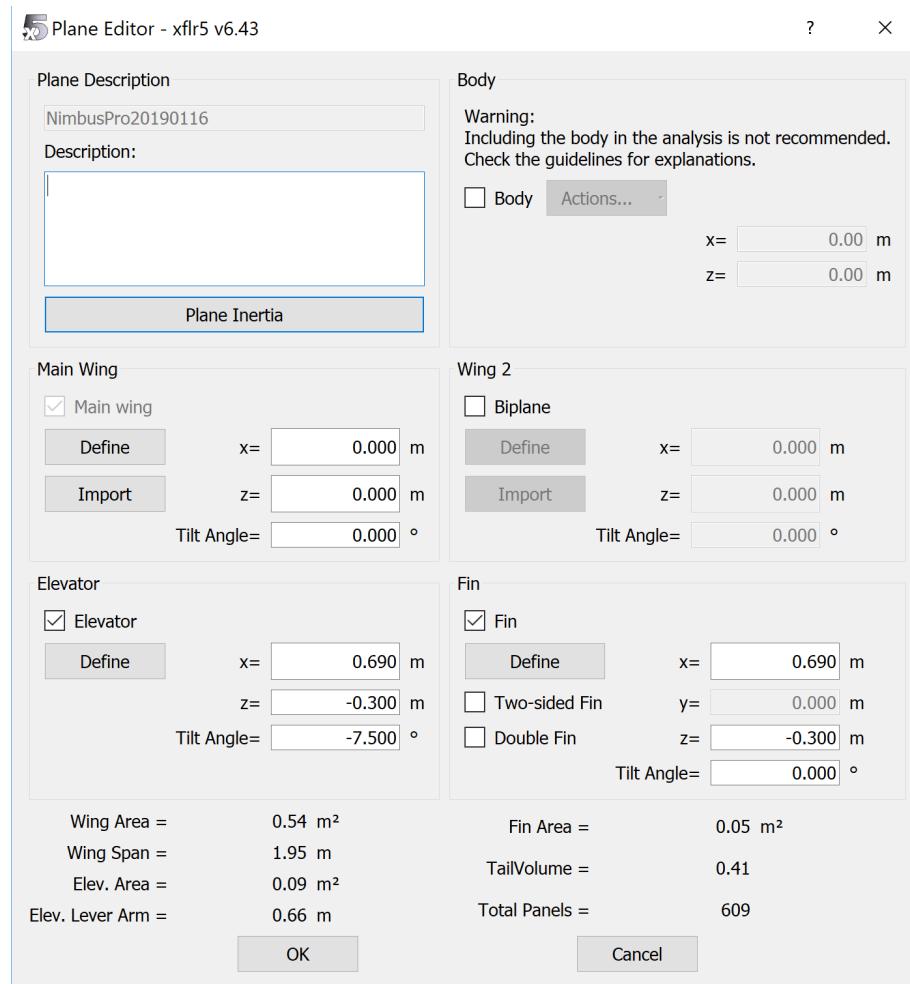


Figure 1: XFLR5 dialog summarizing wing, fin, and elevator parameters.



*Figure 2: Cross section of the wing used to determine its airfoil.*

Finally, the weight of the airframe was determined to be 612 g, the elevator 80 g, and the fin 33 g, distributed uniformly throughout. The remainder of the plane weight was modeled as a point mass of 3095 g, in order to make the total plane weight match the measured 4.1 kg. This could be easily moved in order to change the CG location.

### 3.2 Analysis

Analyses were defined as follows.

#### 3.2.1 Airfoils

First, a batch analysis was performed on the airfoils at the following conditions:

- Mach = 0
- X transition on the upper surface (XtrTop) location: 30%
- Reynold's Number list as defined in Table 3.
- $\alpha$  ranging from  $-8.0^\circ$  to  $18.0^\circ$  at  $0.25^\circ$  increments.
- Other parameters as shown in Fig. 4.

	Re	Mach	NCrit
1	1,000	0.00	9.00
2	3,391	0.00	9.00
3	10,000	0.00	9.00
4	25,000	0.00	9.00
5	50,000	0.00	9.00
6	100,000	0.00	9.00
7	200,000	0.00	9.00
8	225,000	0.00	9.00
9	250,195	0.00	9.00
10	250,781	0.00	9.00
11	251,563	0.00	9.00
12	253,125	0.00	9.00
13	275,000	0.00	9.00
14	300,000	0.00	9.00
15	400,000	0.00	9.00
16	500,000	0.00	9.00
17	750,000	0.00	9.00
18	1,000,000	0.00	9.00

Figure 3: List of Reynold's Numbers used for airfoil analysis.

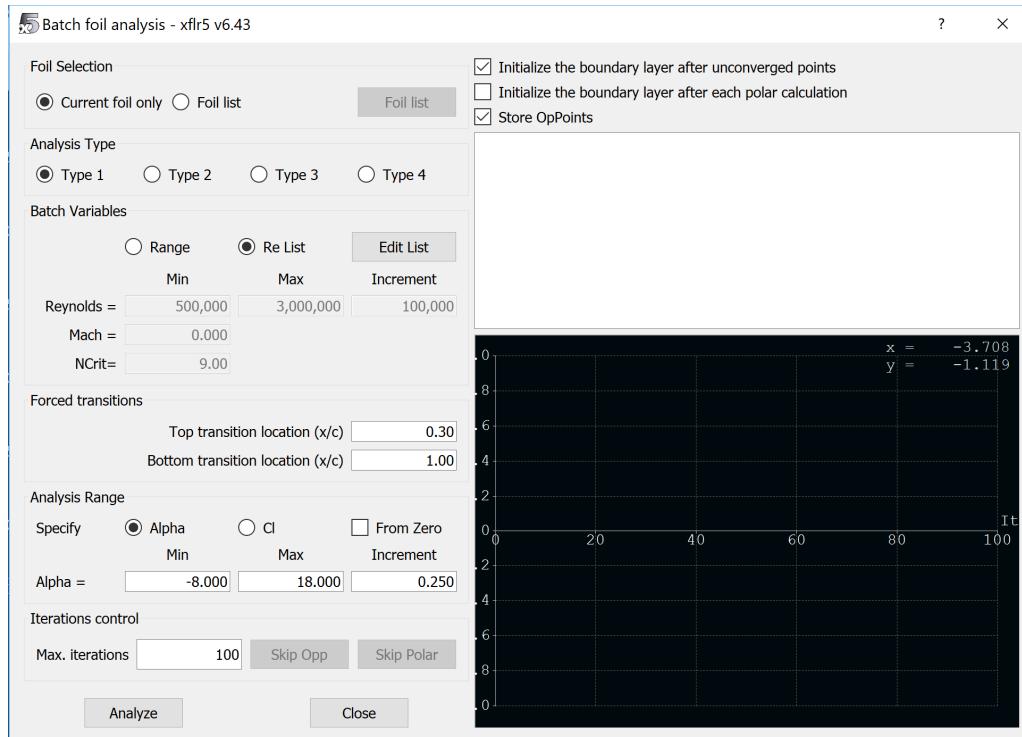


Figure 4: Dialog window for performing an airfoil batch analysis.

### 3.2.2 Plane Analysis

Next, an analysis was defined for the plane as follows. Bold-faced headings correspond to different tabs in the dialog window when an analysis is defined.

#### Polar Type

- Type 2 (Fixed Lift)
- No sideslip ( $\beta = 0$ )

#### Analysis

- Horseshoe vortex (VLM1) (No sideslip)
- Viscous: YES
- Tilt. Geom.: NO

#### Inertia

- Use plane inertia: YES

#### Ref. dimensions

- Wing Planform projected on xy plane

#### Aero data

- $\rho = 1.225 \text{ kg/m}^3$
- $\nu = 1.5e - 05 \text{ m}^2/\text{s}$
- Ground Effect: NO

#### Extra Drag

- Extra area( $\text{m}^2$ ): 0.05
- Extra drag coef.: 0.5\*

\*Note that the drag was roughly approximated as a sphere, with  $C_D = 0.5$ .

#### Analysis settings

- Sequence: YES
- Start =  $-5.000^\circ$
- End =  $15.000^\circ$

### 3.2.3 Stability Analysis

Next, an analysis was defined for the plane as follows. Bold-faced headings correspond to different tabs in the dialog window when a stability analysis is defined.

#### Analysis

- Plane analysis methods: Mix 3D Panels/VLM2
- Viscous Analysis: YES
- $\beta = 0.00^\circ$
- $\phi = 0.00^\circ$

#### Ref. dimensions

- Wing Planform projected on xy plane

#### Mass and inertia

- Use plane inertia: YES

#### Control parameters

- ALL ZERO

#### Aero data

- $\rho = 1.225 \text{ kg/m}^3$
- $\nu = 1.5e - 05 \text{ m}^2/\text{s}$
- Ground Effect: NO

#### Extra Drag

- Extra area( $\text{m}^2$ ): 0.05
- Extra drag coef.: 0.5\*

#### Analysis settings

- Sequence: NO

## 4 Results

The following sections were considered when designing CG placement and design speed.

## 4.1 Airfoils

Drag polars for the airfoil of the main wing at design speed conditions are included in Fig. 5. The design speed condition was selected as  $12m/s$  with a Reynold's number of 224,000 (based on  $\nu = 1.5e - 5$  and a mean aerodynamic chord of  $mac = 0.28 m$ ). Since the linear regime of the  $c_l vs. \alpha$  plot ends at approximately  $9.2^\circ$  and reaches a maximum at approximately  $13^\circ$ . Since we have about  $5^\circ$  of wash-in in the wing design, it is desirable to fly at an angle of attack of  $\alpha < 13^\circ - 5^\circ = 8^\circ$  to avoid tip stall. However, even some tip stall should not be catastrophic as the ailerons lie completely in a 0-twist section of the wing.

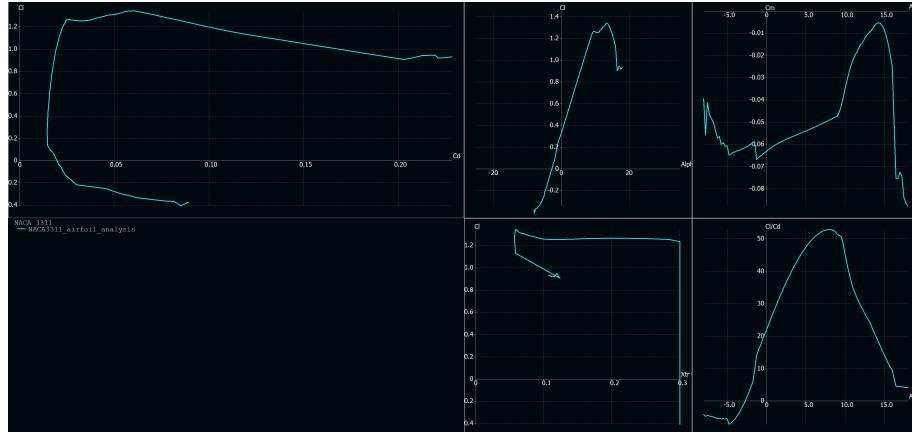


Figure 5: Drag polars of the main wing airfoil at design speed conditions.

## 4.2 Plane Analysis

A rendering of the analyzed plane is included in Fig. 6, with drag polars included in Fig. 7. It is clear from the  $C_L$  vs.  $V$  plot that a lower flight speed corresponds to a higher lift coefficient, with diminishing returns. The lift coefficient varies quite linearly with angle of attack, as shown in the  $C_L$  vs.  $\alpha$  plot. The goal, then, is to increase the design angle of attack as much as possible without running the danger of tip stall. Serendipitously, this has the effect of increasing the lift-to-drag ratio, as seen in the  $C_L/C_D$  vs.  $\alpha$  plot. This will correspond to an increase in range and endurance, both of which are desirable for increasing flight time on a single battery during testing and in the competition. A target design angle of attack of  $7^\circ$  was selected, about  $1^\circ$  away from the tip stall condition discussed in the previous section. Again, since the ailerons are not located in the tip-stall region, we felt comfortable with the small safety factor this entails.

In order to change the design angle of attack, the center of gravity and tail incidence angle were adjusted. The tail incidence angle of the Nimbus Pro is zero off the shelf, requiring significant elevator deflection for steady level flight, increasing required trim and therefore

drag. Further, the XFLR5 analysis could not be performed because control surfaces were not modeled. In order to mitigate this, the tail incidence angle was adjusted in simulation. After placing the CG at an initial guess, the tail angle was increased until a reasonably negative  $C_{m,\alpha}$  (slope of the  $C_m$  vs.  $\alpha$  plot) was obtained. Because the  $C_m$  vs.  $\alpha$  plot also depends on the CG, tail incidence angle and CG were adjusted iteratively. Fig. ?? shows the drag polars resulting from 5 adjustments of the tail incidence angle as well as its . The effect of the tail incidence angle is most apparent in the  $C_{m,\alpha}$  plot (upper right). Note that the bottom-most white curve represents the performance of the plane with its  $0^\circ$  out-of-the-box tail incidence. This is unacceptable without trim, as it would fly at a negative angle of attack, resulting in a near-zero lift coefficient, and a probable crash. Contrast this with the blue curve of the  $C_{m,\alpha}$  plot, which would result in a statically stable flight at an angle of attack of our target  $7^\circ$ .

Next, the center of gravity location was considered. Since steady level flight occurs when the sum of longitudinal pitching moments equals 0, or  $C_m = 0$  on the  $C_m$  vs.  $\alpha$  plot, adjusting the lever arm of the plane's weight has a strong influence on the stable angle of attack. After iterating on the CG location in the plane and tail incidence angle to maximize the lift-to-drag ratio (shown on the  $C_L$  vs.  $C_D$  plot) and minimize design velocity (shown on the  $C_L$  vs.  $V$  plot), it was determined that a CG placement of 6.2 cm in front of the leading edge of the wing results is the optimal condition. The resulting tail incidence angle was  $7.5^\circ$ . This resulted in a design angle of attack of about  $7^\circ$ , consistent with our target value. We calculated a static margin of about 5%. Though slightly low, the plane was still stable, and we believed that the increased aerodynamic performance to be worth the small static margin. Also of note, plans were subsequently made to modify the tail incidence angle to  $7.5^\circ$ . See AF-10 for the tail modification design artifact.

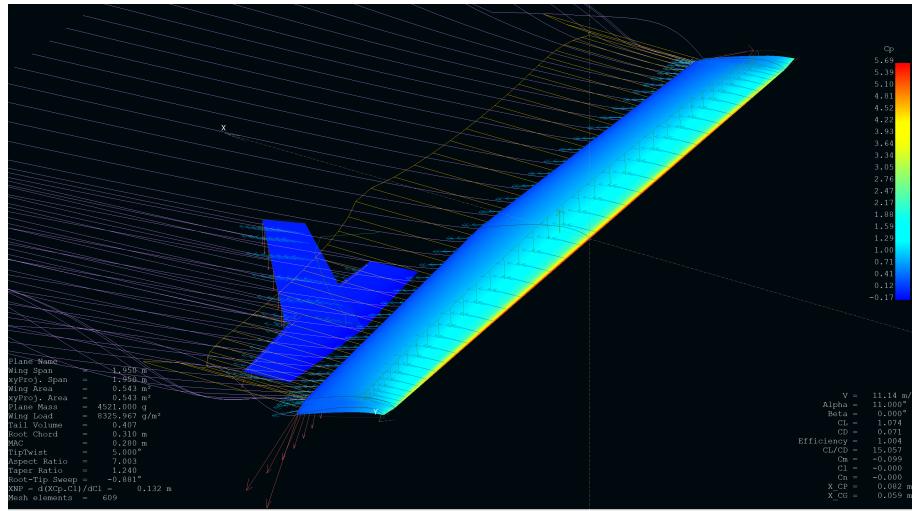


Figure 6: Rendering of the analyzed plane.

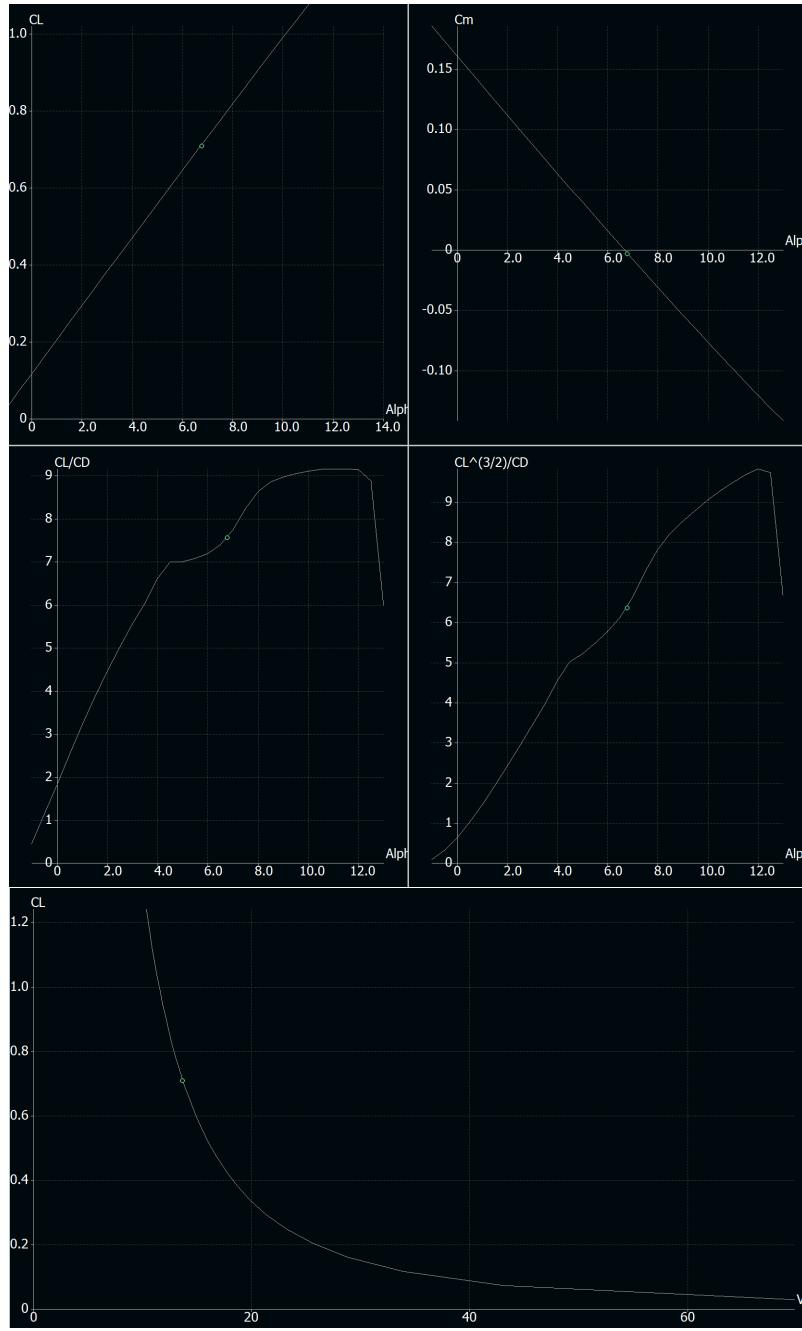


Figure 7: Drag polars of the plane.

## 4.3 Stability Analysis

For increased image clarity and easier waypoint navigation, a statically and dynamically stable airframe is desired.

### 4.3.1 Static Stability

The results of the static stability analysis are included in Fig. 8. Of note:

- $C_{nb} > 0$  indicating static directional stability (yaw axis).
- $C_{ma} < 0$  indicating static longitudinal stability (pitch axis).
- $C_{lb} > 0$  indicating slight static roll instability.

While the roll instability is alarming at first, it is very slight and was believed to be easily overcome by the R/C pilot or the flight controller as needed. Extensive field testing proved the plane to be nearly neutrally roll stable, meaning the plane maintains a constant roll state until ailerons are activated. It is possible that deflection of the foam wings during flight produces a slight dihedral that overcomes any roll instability. the plane may have.

Longitudinal derivatives			
Xu=	-0.73637	Cxu=	-0.16181
Xw=	1.2297	Cxa=	0.27021
Zu=	-6.4391	Czu=	0.0095252
Zw=	-22.465	CLa=	4.9364
Zq=	-5.0124	CLq=	7.8772
Mu=	0.0054678	Cmu=	0.0042964
Mw=	-1.6196	Cma=	-1.2727
Mq=	-1.4419	Cmq=	-8.1034
Neutral Point position=		0.13093 m	

Lateral derivatives			
Yv=	-1.3727	CYb=	-0.30163
Yp=	0.13608	CYp=	0.030667
Yr=	1.0783	CYr=	0.243
Lv=	0.5353	Clb=	0.06032
Lp=	-4.2319	Clp=	-0.48909
Lr=	1.2723	Clr=	0.14704
Nv=	0.92754	Cnb=	0.10452
Np=	-1.0104	Cnp=	-0.11678
Nr=	-0.68476	Cnr=	-0.07914

Figure 8: Tabulated static stability derivatives.

## 4.4 Dynamic Stability

The dynamic stability analysis (included in Fig. 9) reveals negative real parts for all stability eigenvalues except for spiral stability, indicating a stable plane for all but that mode. The severity of this instability was investigated by animating the plane. The "Lateral" radio button was selected and the fourth mode selected in the stability dialog window. The timeframe required for the plane to deflect a single wingspan was nearly 10 seconds; this is ample time for a course correction to be made either by the R/C pilot or the flight controller, and the spiral instability was determined to be negligible.

Longitudinal modes										
Eigenvalue:	-12.03+	-14.81i		-12.03+	14.81i		-0.06469+	-0.9092i		-0.06469+ 0.9092i
Eigenvector:	1+	0i		1+	0i		1+	0i		1+ 0i
	40.47+	22.07i		40.47+	-22.07i		-0.07227+-0.0007608i			-0.07227+0.0007608i
	3.395+	-60.06i		3.395+	60.06i		0.08502+ 0.004928i			0.08502+-0.004928i
	2.332+	2.122i		2.332+	-2.122i		-0.01201+ 0.09266i			-0.01201+-0.09266i

Lateral modes										
Eigenvalue:	-19.85+	0i		-2.499+	-6.355i		-2.499+	6.355i		0.3187+ 0i
Eigenvector:	1+	0i		1+	0i		1+	0i		1+ 0i
	5.269+	0i		0.1333+ 0.1996i			0.1333+ -0.1996i			0.3479+ 0i
	1.272+	0i		0.1385+ 0.4785i			0.1385+ -0.4785i			0.7511+ 0i
	-0.2654+	0i		-0.03435+ 0.00747i			-0.03435+ -0.00747i			1.092+ 0i

Figure 9: Tabulated dynamic stability eigenvalues.

## 5 Conclusion

In summary, the aerodynamic performance of the aircraft was modeled and optimized using the XFLR5 open-source software. The following key results were achieved:

- It was determined that the tail incidence angle should be increased to 7.5° to significantly reduce elevator trim.
- It was determined that the center of gravity should be placed at 6.2 cm from the leading edge of the wing to optimize aerodynamic performance.
- Design speed was decreased from about 20 m/s to 13 m/s: a 35% decrease from last year.
- The design tested sufficiently stable in all modes, static and dynamic.

These test results have provided invaluable direction and insight into the design of this year's airframe.



BRIGHAM YOUNG UNIVERSITY  
AUVSI CAPSTONE TEAM (TEAM 45)

---

## Tail Wedge Design, Construction, and Validation

---

ID	Rev.	Date	Description	Author	Checked By
AF-009	0.1	2-20-2019	Created	Kameron Eves	Tyler Critchfield
AF-009	0.2	4-05-2019	System Refinement edits	Tyler Critchfield	Kameron Eves

## Introduction

During our XFLR analysis detailed in AF-011 we determined that the aircraft would need a significant (-7.5°) horizontal stabilizer incidence angle. The negative incidence angle is very normal in aircraft design and is designed to provide negative lift to push the tail down until the aircraft reaches a trimmed state (i.e. the pitching moment induced by the wings is equal and opposite to the moment induced by the negative lift from the tail). However, the aircraft, as built by the company we purchased it from, has a 0° incidence angle.

The aircraft can still fly in this configuration; the necessary negative lift is caused by deflecting the elevator up (conventionally negative deflection). This effectively causes the horizontal stabilizer to have decreased (moves from 0 towards negative infinity) camber and angle of attack. This induces a negative lift that trims the aircraft. However, this is not an ideal configuration as it necessitates flying with constant deflection on the elevator. Because the equilibrium point for the elevator is not at zero, the throw of the elevator is decreased. In other words, the elevator's saturation point is too close to the equilibrium point and we have decreased control authority. Our first couple flight tests validated this analysis as significant negative elevator deflection was necessary to trim the aircraft. Thus we sought another way to induce this negative lift.

## Design

It was determined that the most effective way to induce this negative lift was to create a wedge that would be mounted between the empennage and the tail of the aircraft. This would give us the required negative incidence angle, but would also decrease the sweep on the vertical stabilizer and adjust the rudder so that it is not exactly orthogonal to the aircraft body. We decided, however, that these were acceptable trade-offs as leading edge sweep only has a positive effect on the aerodynamics (decreasing the drag) if the airspeed of the aircraft is near the speed of sound. Because our design flight speed is significantly below the speed of sound (15 m/s), we decided that decreasing the leading edge sweep of the vertical stabilizer is acceptable.

Angling the rudder would have adverse consequences on its effectiveness and cause coupling between the rudder and the pitching moments. However, the developers of the autopilot we are using have found that the rudder provides very little benefit to the aircraft over all control schemes. The rudder is usually used to mitigate sideslip and allow for coordinated turns, but the sideslip is usually mitigated sufficiently by having a large horizontal stabilizer (which we do) and coordinated turns are only really beneficial to passenger comfort (something we don't care about as this aircraft will not be carrying passengers). Thus the autopilot we use commands the rudder to be constant at its trim

value (zero or very close to zero) and adjustments to the rudder mounting orientation should have little to no effect on our overall flight characteristics.

From the above analysis we decided that a wedge would be mounted between the tail and the empennage. This wedge needed to have a flat bottom to allow it to be glued to the applicable section of the tail and another flat surface  $7.5^\circ$  from the bottom for the tail assembly to be attached to it. Finally, the outer profile of the wedge only needed to match the outer profile of the tail to reduce drag. Artifact AF-010 is a dimensioned CAD drawing showing the wedge shape. Important to note is that the outer profile of the wedge is not dimensioned. The outer shape of the wedge follows the complex profile of the aircraft's tail. For more exact specification, contact the manufacturer of the Nimbus Pro aircraft. See the construction shape below for a description of how we manufactured this shape.

Pink insulation foam that can be purchased at any hardware store was chosen as the material for the wedge because it was readily available, cheap, light-weight, and would not have to bear significant forces (the large moment arm of the tail means that the forces necessary to trim the aircraft are small and the wide base of the wedge means that the stresses induced by those forces are even smaller).

## Construction

The wedge's outer profile was cut to the desired angle by a foam cutter. The raw material was placed in the foam cutter and the custom shape was loaded into the cutting software. The profile cut by the foam cutter only gave the desired angle for the wedge. The outer profile of the wedge was cut by holding the wedge in the position it was to be installed in and tracing the geometry of the tail assembly directly on to the cut wedge. This was then cut again using the foam cutter's heated wire, but this time we manually manipulated the foam to produce the required shape. Light-weight RC aircraft grade glue was used to bind the wedge to the tail and empennage. Figure 1 shows the final product.

## Testing & Validation

Upon finishing construction we performed a flight test and trimmed the elevator in the new configuration. We were pleased to find that our predictions and analysis very closely matched the actual result as we needed zero deflection to trim the aircraft in this new configuration. Figure 2 shows the empennage after the flight where we trimmed the aircraft.



*Figure 1: The wedge finished and installed on the aircraft. Note that the outer surface of the wedge is flush with the outer surface of the aircraft’s tail. It should be noted that the orientation of this image makes the wedge appear larger than it actually is—the angle is still 7.5°.*

## Conclusion

We are very pleased with the implementation of this wedge as its success will aid in meeting our requirements and key success measures. Primarily, the wedge will mitigate the risk of saturating our elevator and thus crashing because we do not have the control authority to recover. This benefit has been captured in the failure modes and effects analysis that can be found in AF-007. Mitigating the chance of failure will allow all of our key success measures to be achieved. More specifically, the increased control authority will allow for increased precision in our flight path. This will help us decrease the number of obstacles hit and increase how close we get to waypoints.



*Figure 2: The empennage after trimming the aircraft. As can be seen the deflection on the elevators is 0°, indicating that the wedge causes the correct incidence angle and negative lift.*



BRIGHAM YOUNG UNIVERSITY  
AUFSI CAPSTONE TEAM (TEAM 45)

---

## Flight Test Log

---

ID	Rev.	Date	Description	Author	Checked By
AF-004	0.1	11-07-2018	Created*	Kameron Eves	Andrew Torgesen
AF-004	0.2	2-5-2019	Added tracking of autonomous and total flight time	Kameron Eves	Andrew Torgesen
AF-004	1.0	2-5-2019	Added Take-off/Landing Tracker	Kameron Eves	Brandon McBride
AF-004	2.0	4-5-2019	Adjusted format of totals	Kameron Eves	Andrew Torgesen



## Flight Test Log

\*Note that additions to this log will not necessitate a revision update. Only formatting or other content changes will require that.

## Log

*Table 1: A log of each flight test conducted by our team. Autonomous flight time is listed in bold under the total time.*

Date (m-d-y)	Location	Length Total <b>Auto</b> (min)	Takeoffs/ Landings <b>Auto</b>	Notes
10-16-18	Springville	<1	1/0	Networking issues, later determined to be because of location. Moving down the road works. Attempted RC flight and crashed on launch. Need to practice launch procedure.
10-19-18	Springville	<1	1/0	Attempted RC flight for imaging. RC lost upon launch. Later determined to be because of the RC antenna not being installed. Aircraft did not have a balanced center of gravity (CG) and so performed a loop and crash landed.
10-23-18	Springville	1	1/0	Attempted RC flight for imaging. Aircraft had major longitudinal stability issues. Later determined to be because of a negative static margin. Moving the battery forward fixes issue. Lost control and crashed. Transmitter also dying very very quickly. Later determined to be transmission power set to high (1 A changed to 10 mA).
11-01-18	Springville	3	1/0	Attempted RC flight for imaging. Still had minor stability issues. We lost control and crash landed near end of flight. We later determined these issues were because the battery was not secured properly. It slid around inflight affecting our static margin. This caused instability and aggressive flight maneuvers that caused the battery to fall out in flight. As such we lost control and crashed. Battery must be strapped down.

## Flight Test Log

---

11-06-18	Springville	5	1/0	Attempted RC flight for imaging. Aircraft flew wonderfully. Images of ground targets successfully captured. Flight was terminated when RC was lost and aircraft crashed hard. More investigation into the cause is needed. Possibly because of RC interference over the trees or to low of transmission power.
12-11-18	Rock Canyon	1	1/0	First flight test of new aircraft. Performed a glide test and found the aircraft performed well. With slight longitudinal instability. We moved the cg forward by adding a 500 g weight. This proved too much as the aircraft was notably nose heavy the next flight attempt which was forced to landed immediately upon takeoff . With an inexperienced pilot, we found this preferable to instability and attempted a second flight. The aircraft still dropped on takeoff but stable flight was obtained. This lasted 45 seconds before the pilot lost control and crashed. While pilot error likely played a part in this crash, we later determined that wind from the canyon was a large factor.
1-11-19	Rock Canyon	6	1/1	First fully successful flight. Set trims. We also decided to add colored tape to the wings to increase the visibility of the aircraft in flight.
1-18-19	Rock Canyon	4	1/1	Very windy, probably shouldn't have attempted flight but we had not yet figured out the wind problem from the canyon. Adjusted trims.

## Flight Test Log

---

1-23-19	Rock Canyon	14	2/2	Performed two flights both of which were successful. Several minor repairs (general maintenance) were necessary after this flight. This flight test proved the aircraft was flyable and stable with all of the weight that will be on the aircraft during the competition. We tested both the weight with and without the UGV.
1-25-19	Rock Canyon	11	2/2	Trimmed aircraft with and without UGV weight. Transferred these trims to ROSflight. 2 flights. Additional tape on wing provided sufficient visibility.
1-31-19	Utah County Airfield	7	1/1	Intended to test autopilot and begin tuning gains. Could not successfully turn on autopilot. Later this was determined to be because we were incorrectly following the process to hand over control to the autopilot. We flew once manually to test the gains. Small adjustments were made and transferred to ROSflight.
2-2-19	Utah County Airfield	24 1	2/2	Two flights to tune gains on autopilot. The autopilot had a tendency to flip the aircraft upside down immediately after turning on the autopilot. This was determined to not be caused by the gains. Aircraft landed safely manually both flights. After a couple of days of testing we found the cause was that the number being used to convert rad to PWM for the ailerons was negative. This effectively reversed the aileron polarity. Last year the wires must have been swapped from how we have them now.

## Flight Test Log

---

2-6-19	Utah County Airfield	<b>17 4</b>	2/2	Two flights to tune gains on autopilot. Flipping issue was fixed. Aircraft dove towards the ground upon turning on autopilot. This was fixed between flights (rad to PWM conversion for the elevator was negative this time) and we achieved autonomous flight the second attempt. We then tuned the longitudinal PID gains. Aircraft landed safely manually both flights.
2-6-19	Utah County Airfield	<b>31 10</b>	1/1	Tuned longitudinal gains and made a small effort to tune lateral gains. We also attempted a loiter, but aircraft was not tuned well enough to do so. Used course following to perfect the gains. Finished the longitudinal gains and got lateral gains reasonable. Next step is attempting a loiter and waypoints to tune lateral gains.
2-19-19	Rock Canyon	3	1/1	Short flight to test cargo drop. Payload dropped upon command and the parachute opened successfully.
2-26-19	Rock Canyon	10	3/3	Three more flights to test payload drop. Payload successfully dropped two out of the three times. During the second test, the payload door got stuck on some tape we added between flights and the payload didn't fall out until a full minute after it was commanded. We removed this tape and the aircraft performed nominally on the third flight. We also used this flight to get realistic images to test the vision algorithm on. The camera was installed and images save on the odroid.

## Flight Test Log

---

3-5-19	Utah County Airfield	10	1/1	Miss communication on what code was on the aircraft inhibited our ability to test the control algorithms. As such we will put in place protocol for changing code on the aircraft (see AF-014) Aquired imaging data.
3-25-19	Utah County Airfield	0	0/0	During preflight check we experienced RC brown outs. Other RC pilots at the airfield experienced a similar problem, which caused us to think interference might be the cause. We did not risk a flight. In the lab, we were able to replicate the problem. It seemed to be caused by a poor wire connection between the RC transmitter and the amplification packet attached to it. Further discussion with the other RC pilots indicated that they thought their issue had to do with a loss of propulsion. These two facts seem to indicate that it was not actually an interference issue, but instead a hardware issue. We fixed the loose connection and it seemed to solve the problem.
3-27-19	Utah County Airfield	28 <b>9</b>	3/2 <b>0/1</b>	Three flights. Dropped payload successfully twice. Flew 3 waypoints and landed autonomously.
4-1-19	Utah County Airfield	4	1/0	Safety pilot accidentally disarmed the plane while trying to transition to the autopilot. This caused loss of power, and, after this was realized, it was too late to save the aircraft from crashing. Aircraft landed in a bush and was mostly saved.

## Flight Test Log

---

4-2-19	Utah County Airfield & Rock Canyon	0	0/0	During preflight checks we could not achieve wireless communication between the flight computer and ground station. We found that the Ubiquity bullet never connected to the light beam antenna. We spent over an hour attempting to solve the problem. We tried moving the aircraft a few feet (which had worked in the past), replacing wires that might have been bad, and scanning for interference. We rebooted the system after each attempt to fix the problem. Nothing worked. Eventually we gave up and left. On the drive back we decided to try again at Rock Canyon park. Of course, at Rock Canyon, the wireless signal connected on the first try. Issues such as this were observed earlier in the year, but are not repeatable and occur so infrequently that we can not get to the root of the problem. We still have no clue what the issue is or how to solve it. It seems to be simultaneously location and time dependent.
--------	--	---	-----	---

4-3-19	Rock Canyon	10 2	1/0	Experienced RC signal brown outs. We didn't notice this until after we had started flying. Small amounts of autonomous flight were achieved. Because the RC signal dropped out, the aircraft entered a loiter successfully (it's failure mode). The safety pilot then tried to recover the aircraft and, because RC signal was spotty, could not control the aircraft. This caused a crash in a tree. The aircraft was still mostly flyable, but we could not find the bug in the RC signal. It seems to occur randomly. While we did get it to occur on the ground once, the problem in general is not intentionally repeatable and does not occur frequently enough for easy debugging. We have some guesses to what the issue is (again poor connection between the transmitter and amplification packet), so are currently using the guess and check method to try to solve this issue.
--------	-------------	---------	-----	---

---

### Statistics

---

**Total Flight Time:** 3 Hour 9 Minutes

**Manual Flight Time:** 2 Hour 43 Minutes\*

**Autonomous Flight Time:** 26 Minutes

**Percent of Autonomous Flight:** 13.8%

---

**Manual Takeoffs:** 29\*

**Manual Landings:** 20\*

**Autonomous Takeoffs:** 0

**Autonomous Landings:** 1

---

\*With the aircraft configuration and safety pilot to be used in the competition



BRIGHAM YOUNG UNIVERSITY  
AUVSI CAPSTONE TEAM (TEAM 45)

---

## Field Flight Checklist v2.0

---

ID	Rev.	Date	Description	Author	Checked By
PF-001	0.1	11-03-2018	Initial creation	Andrew Torgesen	Brandon McBride
PF-001	0.2	01-07-2019	Updates from feedback	Andrew Torgesen	Tyler Miller
PF-001	1.0	02-04-2019	Updates from feedback	Andrew Torgesen	Kameron Eves
PF-001	2.0	03-06-2019	Updates from feedback	Andrew Torgesen	John Akagi

## 1 Purpose

The purpose of this artifact is to keep an up-to-date, standard protocol for ensuring safety and good performance for test flights in hardware. It is important that all test flights are run systematically, and according to the procedures and timelines outlined in this document.

## 2 Checklist

### Day Before

- Check that the launch file does what it needs to with the plane grounded
- Ensure that the ROSbag records the data you want
- Charge airplane LiPo(s)
- Charge RC transmitter battery
- Parameter check
- Check WiFi config
- Check disk space on Odroid

---

### Hardware Packing List

- Plane
- Wings
- Airplane batteries
- RC transmitter
- RC transmitter batteries
- 2+ sets of props
- Fiber tape
- Launch gloves
- Wrench for props
- Pliers

- 
- Battery monitor
  - Safety glasses
  - Screwdriver
  - Table (optional)
  - Targets (optional)
- 

### Comms Packing List

- Router + power cable
  - Litebeam + 2 ethernet cables
  - A/C POE adapter
  - Extra ethernet cable
  - Car power adapter
  - 3-plug extension cable
  - Walkie-talkies
  - Generator (optional)
- 

### Flight Checklist: *Before Launching*

Before Powering Motor:

- Start network
- Attach wings
- Attach props and check tightness
- Strap down battery
- Connect battery monitor (full battery: 16.8 V)
- Check plane CG
- Turn on transmitter
- Ensure that the arm and RC override channels are set to OFF
- (TEMPORARY) Hold UGV latch shut while connecting battery

- Connect battery
- (TEMPORARY) Release UGV latch when you hear it latch twice
- Ensure network connection
- Launch ROS (through *screen*, if possible) (ensure aircraft is level)
- Ensure GPS Fix ( $\geq 3$  satellites)
- Calibrate Sensors
  - IMU: rosservice call */calibrate\_imu*
  - Airspeed: rosservice call */calibrate\_airspeed*
  - Barometer: rosservice call */calibrate\_baro*
  - Check attitude estimation (except for yaw—if wrong, update ins offset)
  - Check airspeed
  - Check GPS
- Check RC
  - Ensure RC transmitter is emitting enough power ( $> 10 \text{ mW}$ , 1 W in competition)
  - Wire wiggle test
  - Check control surface direction
    - Ailerons
    - Elevators
  - RC Range Test (100ft, just do this once per setting config change)
- Lock shut hatch covers
- Check Autopilot
  1. Begin with throttle 0%, Arm OFF, RC Override ON (both top switches toward the pilot)
  2. ROStopic echo */status*
  3. Secure aircraft (hold firmly)
  4. Arm ON
- Confirm *armed = true*

5. RC Override OFF
6. Perform the following in quick succession (no longer then 2 seconds)
  - (a) Call "Clear Props"
  - (b) Throttle to full
    - Confirm *RC Override = false*
    - Confirm air blowing towards tail
  - (c) Throttle to idle
    - Confirm prop direction

---

**FLY**

- Takeoff
    - Ensure area clear
    - Get into position
    - Go/No Go Call
      - Vision
      - UGV
      - Autopilot
      - Antenna Pointer
      - RC Pilot
      - Launcher
      - Team lead
  - Arm ON
  - RC Override OFF
  - Throttle full
  - Toss the aircraft
- RC Takeover
    - RC Override ON

- 
- Throttle to desired
  - Handover to Autopilot
    - RC Override OFF
    - Throttle to full
- 

#### Flight Checklist: *After Landing*

- Kill ROS
  - Backup ROSbag
  - Clean shutdown
  - Unplug battery
  - Gather all items
- 

#### Post-flight

- Set battery to storage voltage



BRIGHAM YOUNG UNIVERSITY  
AUVSI CAPSTONE TEAM (TEAM 45)

---

## Imaging Subsystem Description

---

ID	Rev.	Date	Description	Author	Checked By
IM-005	1.0	12-11-2018	Initial release	Tyler Miller & Connor Olsen	Derek Knowles
IM-005	1.1	02-14-2019	Update diagrams. Clarification	Tyler Miller	Connor Olsen

## 1 Introduction

Vision's imaging server will be the central interface between the raw data received from the plane the various ODLC (Object Detection Localization Classification) clients. Below is a brief overview of how it is setup and where its various components reside.

## 2 Motivation

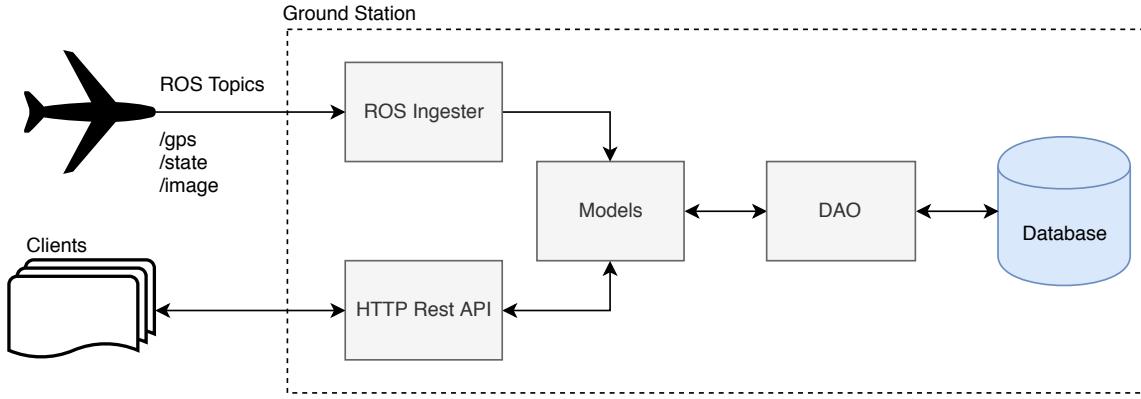
This major code change was primarily motivated by using last years imaging software and observing its shortcomings. Last years software required overly complex and difficult configuration and setup in order to produce actionable manual classifications. Manual classification required 3-4 people, when it could be easily managed by 1 or 2. Finally, the system was fully dependent on ROS and was interconnected in such a way that it would be next to impossible to change a single module without the entire package being affected.

Given these shortcomings we set out to create an imaging package that emphasized the following:

1. Support up to N manual and autonomous imaging clients out of the box with no special configuration required.
2. Maximize modularity and transferability by separating the platform from ROS as much as possible.
3. Provide extensive documentation to ease future development and enable new developers to learn quickly.
4. Build and release the code base in such a way that it can be run using a single command on any platform.

## 3 Server

Once we fully understood last years imaging system we were able to lay out its weaknesses and create the above objectives to best address them. One of our main concerns was how the current codebase spread its information across multiple machines: one machine would hold the raw images from the plane's camera, another would hold cropped images, and still another would hold classification information. This made it very difficult to identify bugs, replicate results and setup quickly. With these objectives and shortcomings in mind, a basic server-client architecture was defined as shown in Figure 1.



*Figure 1: Server Architecture*

Notice that the ROS Ingester is the only code block dependent on ROS. In reality, the ingester is a simple 200 line python script which subscribes to the relavent ROS data streams (aka: rostopics), and pushes their information into the database.

Model classes represent a single row in a table and essentially function as a translation layer between the database and the top level ingester/Rest API.

DAO's or Database Access Objects, interact directly with the database by encapsulating SQL queries into methods that can be easily accessed and called by external modules. These methods return model classes or require them as parameters.

For the database we chose Postgresql. Postgres is well known for its powerful feature set, as well as easy out-of-the-box concurrency safety. The later was particularly important in our application since there will often be multiple connections simultaneously querying most of the database.

Rest APIs are a ubiquitous interface and fit Vision's problem space well. By publishing a Rest server and building our package around its ideology, support for up to N clients comes naturally.

The basic data flow of all these components is shown in Figure 2.

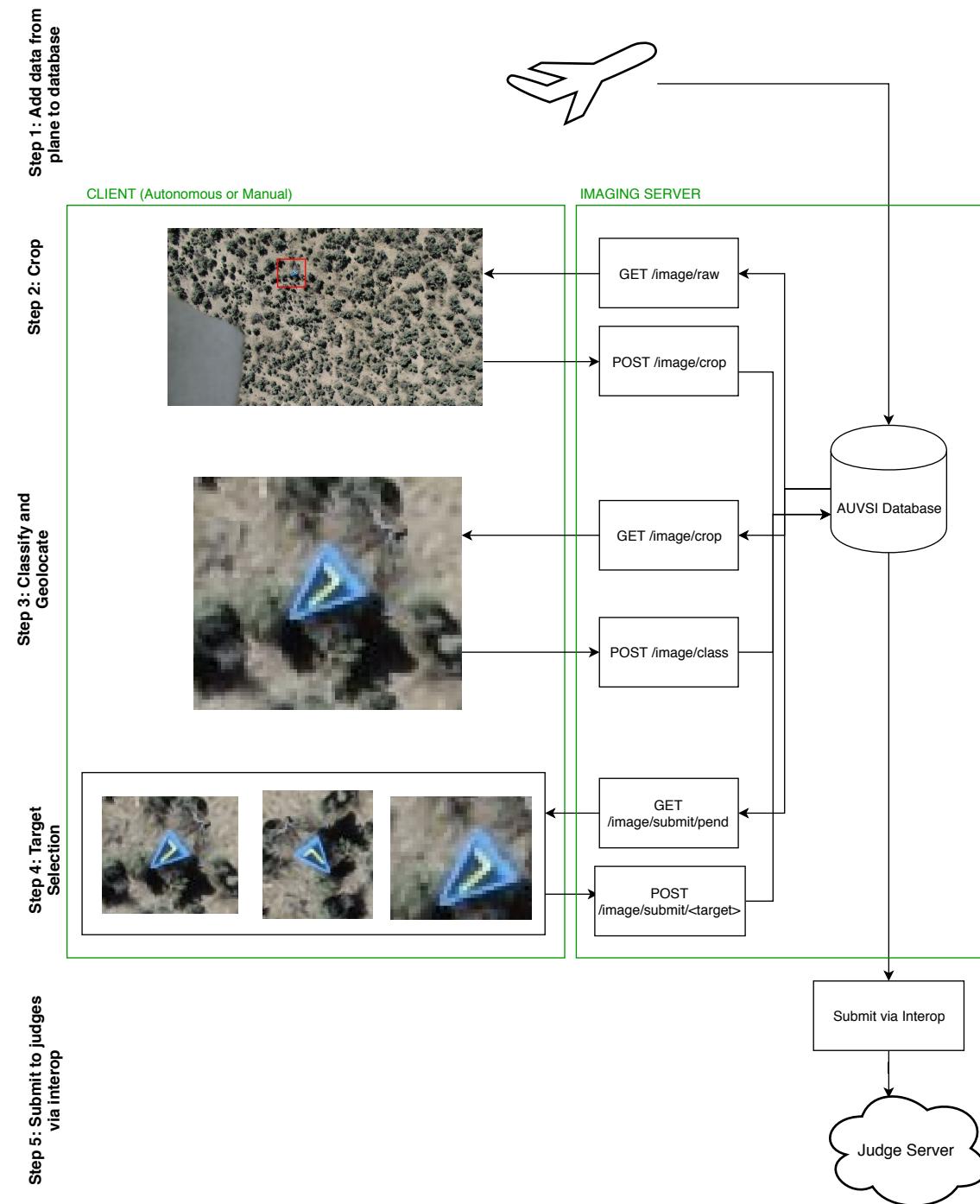


Figure 2: AUVSI Imaging Data Flow Through Autonomous/Manual Classification

Data flows back and forth between client and server, with the server holding a definitive-final copy of a target image, as well as a history of its state during intermediate steps.

**Step 1** Takes all the data from the plane and pushes it into the database on the server. This is the "raw" data collected by the plane, which the server will store for the duration of the flight.

**Step 2** An autonomous or manual client requests the next raw image from the server. The server automatically keeps track of which images a client has or has not seen and will only send new photos. The client then attempts to locate any targets, and will POST a cropped image to the server if it does.

**Step 3** An autonomous or manual client requests an unclassified cropped image. They can then proceed to classify it, identifying attributes such as shape, letter, and color. All of these attributes are required by the AUVSI judges for maximum points. At this point the cropped image is also automatically run through a geolocation script (detailed more in IM-008) which attempts to pinpoint the targets latitude and longitude.

**Step 4** Often the plane will capture multiple images of the target, and thus multiple cropped images and classifications for that target will be generated by the previous steps. The server automatically bins similar classifications as a single 'target'. In this step, users can choose which image and classification details will finally be submitted to the judges (since we want to only submit a target once).

**Step 5** The server gets any targets declared ready for submission and sends them to the interop subsystem. Interop is in charge of interactions between the judge server and all subsystems and will handle submitting the final targets.

The manual client GUI codebase is detailed in depth in IM-001 and IM-002. Details on the geolocation algorithm can be found in IM-008, while a description of the server's API can be found in IM-003.

## 4 Conclusion

Properly implemented, the server infrastructure allows simple one click installation for anyone looking to run imaging. Coupled with the vision team's emphasis on comprehensive documentation and official code releases, the imaging codebase will be as transferable as possible for future AUVSI team members.

Besides this high degree of transferability, this design is also modular. Different portions of the codebase could be re-implemented or changed with little to no effect on other code modules. The strengths of the new imaging model respond to some of the largest issues in

the old imaging codebase and create a reliable, easy to use, modular solution to AUVSI's imaging problem



BRIGHAM YOUNG UNIVERSITY  
AUVSI CAPSTONE TEAM (TEAM 45)

---

## Imaging Requirements Matrix

---

ID	Rev.	Date	Description	Author	Checked By
IM-007	1.0	02-20-19	Measured and predicted values	Tyler Miller & Jake Johnson	Connor Olsen

## 1 Introduction

Figure 1 shows our updated requirements matrix for the imaging subsystem. Predicted and measured values, as well as market response were added where applicable. Some values can not be fully measured until multiple flight tests are performed with imaging fully integrated into the plane.

## Imaging Requirements Matrix

Subsystem Performance Measures										Units	Market Response		
Importance													
	1	2	3	4	5	6	7	8	9				
<b>Target Design Requirements</b>													
1 Detect minimum object size manually	9									Images/second	Good		
2 Determine object geolocation within 30 m manually	9									Meters/second	Good		
3 Detect minimum object size autonomously	9									Pixels/in	Neutral		
4 Determine object geolocation within 30 m autonomously	9									Percent	Good		
5 Rate of ground station contact is 10 hz (Interop)	6									Count	Very Good		
6 Differentiate between five distinct shapes	6									Count	Very Good		
7 Differentiate between five distinct colors	6									Percentage	Very Good		
8 Determine alphanumeric characters manually	6									Hertz	Very Good		
9 Determine alphanumeric characters autonomously	6									Count	Neutral		
Measured	Predicted	Upper Acceptable	Ideal	Lower Acceptable									
0.5	N/A	1	0.5										
13	15	12	7										
1.67	N/A	1.1	0.7										
TBD	5	90	30	2									
TBD	10	10	5	3									
TBD	13	13	5	3									
TBD	5	N/A	20	5									
20	N/A	20	10										
TBD	20	26	5	2									



BRIGHAM YOUNG UNIVERSITY  
AUVSI CAPSTONE TEAM (TEAM 45)

---

## Imaging GUI API

---

ID	Rev.	Date	Description	Author	Checked By
IM-002	1.0	12-07-2018	Initial release	Derek Knowles & Brandon McBride	Connor Olsen

## 1 Introduction

The purpose of this artifact is to document the code for the graphical user interface used to crop, classify, and submit target images to the judges server during the AUVSI competition. This document lists a brief description for each function and the functions input and output parameters.

## 2 Module client\_gui

Authors: D. Knowles, B. McBride, T. Miller

Prereqs: python 3 sudo apt install python3-tk pip3 install Pillow, opencv-python, ttkthemes

### 2.1 Class GuiClass

```
tkinter.Frame └─  
    client_gui.GuiClass
```

Graphical User Interface for 2019 AUVSI competition  
Tab 0: Setting for setting up the server\_error  
Tab 1: Pull raw images and submit cropped images  
Tab 2: Pull cropped iamges and submit classification for images  
Tab 3: Display results for manual and autonomous classification

#### 2.1.1 Methods

`__init__(self, master=None)`

initialization for Gui Class

**Parameters**

`tk.Frame`: nothing

*(type=nothing)*

**Return Value**

None

*(type=None)*

`get_image(self, path)`

Reads in an image from folder on computer

**Parameters**

`path`: the file path to where the image is located

*(type=file path)*

**Return Value**

Numpy array of selected image

*(type=Numpy image array)*

**np2im(self, image)**

Converts from numpy array to PIL image

**Parameters**

**image**: Numpy array of selected image  
    (*type=Numpy image array*)

**Return Value**

  PIL image of numpy array  
    (*type=PIL image*)

**im2tk(self, image)**

Converts from PIL image to TK image

**Parameters**

**image**: PIL image of numpy array  
    (*type=PIL image*)

**Return Value**

  TK image of PIL image  
    (*type=TK image*)

**mouse\_click(self, event)**

Saves pixel location of where on the image the mouse clicks

**Parameters**

**event**: mouse event  
    (*type=event*)

**Return Value**

  None  
    (*type=None*)

**mouse\_move(self, event)**

Gets pixel location of where the mouse is moving and show rectangle for crop preview

**Parameters**

**event**: mouse event  
    (*type=event*)

**Return Value**

  None  
    (*type=None*)

**mouse\_release(self, event)**

Saves pixel location of where the mouse clicks and creates crop preview

**Parameters**

event: mouse event  
*(type=event)*

**Return Value**

None  
*(type=None)*

**close\_window(self, event)**

Closes gui safely

**Parameters**

event: ESC event  
*(type=event)*

**Return Value**

None  
*(type=None)*

**resizeEventTab0(self, event=None)**

Resizes picture on Tab0

**Parameters**

event: resize window event  
*(type=event)*

**Return Value**

None  
*(type=None)*

**resizeEventTab1(self, event=None)**

Resizes pictures on Tab1

**Parameters**

event: resize window event  
*(type=event)*

**Return Value**

None  
*(type=None)*

**resizeEventTab2(self, event=None)**

Resizes picture on Tab2

**Parameters**

event: resize window event  
*(type=event)*

**Return Value**

None  
*(type=None)*

**resizeIm(self, image, image\_width, image\_height, width\_restrict, height\_restrict)**

Resizes PIL image according to given bounds

**Parameters**

image: PIL image that you want to crop  
*(type=PIL image)*  
image\_width: the original image width in pixels  
*(type=integer)*  
image\_height: the original image height in pixels  
*(type=integer)*  
width\_restrict: the width in pixels of restricted area  
*(type=integer)*  
height\_restrict: the height in pixels of restricted area  
*(type=integer)*

**Return Value**

Resized PIL image  
*(type=PIL image)*

**cropImage(self, x0, y0, x1, y1)**

Crops raw image

**Parameters**

x0: pixel x location of first click  
*(type=integer)*  
y0: pixel y location of first click  
*(type=integer)*  
x1: pixel x location of second click  
*(type=integer)*  
y1: pixel y location of second click  
*(type=integer)*

**Return Value**

None  
*(type=None)*

**undoCrop(self, event=None)**

Undoes crop and resets the raw image

**Parameters**

event: Ctrl + Z event  
*(type=event)*

**Return Value**

None  
*(type=None)*

**nextRaw(self, event)**

Requests and displays next raw image

**Parameters**

event: Right arrow event  
*(type=event)*

**Return Value**

None  
*(type=None)*

**previousRaw(self, event)**

Requests and displays previous raw image

**Parameters**

event: Left arrow event  
*(type=event)*

**Return Value**

None  
*(type=None)*

**submitCropped(self, event=None)**

Submits cropped image to server

**Parameters**

event: Enter press or button press event  
*(type=event)*

**Return Value**

None  
*(type=None)*

**nextCropped(self, event)**

Requests and displays next cropped image

**Parameters**

event: Right arrow event  
*(type=event)*

**Return Value**

None  
*(type=None)*

**prevCropped(self, event)**

Requests and displays previous cropped image

**Parameters**

event: Left arrow event  
*(type=event)*

**Return Value**

None  
*(type=None)*

**submitClassification(self, event=None)**

Submits classification of image to server

**Parameters**

event: Enter press event  
*(type=event)*

**Return Value**

None  
*(type=None)*

**tabChanged(self, event)**

Performs the correct keybindings when you move to a new tab of the gui

**Parameters**

event: Tab changed event  
*(type=event)*

**Return Value**

None  
*(type=None)*

**updateSettings(*self*, *event=None*)**

Attempts to connect to server when settings are changed

**Parameters**

**event:** Enter press or button press event  
*(type=event)*

**Return Value**

None  
*(type=None)*

**pingServer(*self*)**

Checks if server is correctly connected

**Return Value**

None  
*(type=None)*

**disableEmergentDescription(*self*, \**args*)**

Disables emergent discription unless emergent target selected

**Return Value**

None  
*(type=None)*



BRIGHAM YOUNG UNIVERSITY  
AUVSI CAPSTONE TEAM (TEAM 45)

---

## Imaging GUI-Server Interface API

---

ID	Rev.	Date	Description	Author	Checked By
IM-001	1.0	12-07-2018	Initial release	Brandon McBride & Derek Knowles	Connor Olsen
IM-001	1.1	02-19-2019	Subsystem updates	Tyler Miller	Derek Knowles

## 1 Introduction

The purpose of this artifact is to document the code that interfaces with the GUI and server. This document lists a brief description for each function and the functions input and output parameters.

## 2 Module client\_rest

### 2.1 Functions

<code>testNextAndPrevRawImage(interface)</code>
---

<code>testNextAndPrevCroppedImage(interface)</code>
---

<code>testCropPost(interface, imgId)</code>
---

### 2.2 Variables

Name	Description
<code>--package--</code>	<b>Value:</b> None

### 2.3 Class ImageInfo

An ImageInfo object holds the information concerning an image such as the image id, the image path, the timestamp the image was taken, and whether or not it has been seen by the autonomous or manual system.

### 2.3.1 Methods

```
__init__(self, auto_tap, imageId, path, man_tap, ts)
```

The constructor for the ImageInfo class.

#### Parameters

- `auto_tap`: True if the autonomous system has seen the image, False otherwise  
*(type=boolean)*
- `imageId`: the id related to the image  
*(type=int)*
- `path`: the file path to where the image is located  
*(type=String)*
- `man_tap`: True if the manual system has seen the image, False otherwise  
*(type=boolean)*
- `ts`: the timestamp that the image was taken  
*(type=float)*

#### Return Value

- None  
*(type=None)*

## 2.4 Class CropInfo

A CropInfo object holds information concerning a cropped image such as the image id to the original image, the top-left and bottom-right coordinates to where the crop took place on the original image, the path of the cropped image, whether or not the cropped image has been seen by the imaging system, and the timestamp of the original image.

### 2.4.1 Methods

```
__init__(self, imgId, tl, br, path, isTapped, ts)
```

The constructor for the CropInfo class.

#### Parameters

- imgId:** the id of the original image  
*(type=int)*
- tl:** the x and y coordinates of the original image of the top left corner to where the image was cropped  
*(type=int[])*
- br:** the x and y coordinates of the original image of the bottom right corner to where the image was cropped  
*(type=int[])*
- path:** the file path to where the cropped image is located  
*(type=String)*
- isTapped:** True if the manual imaging system has seen this image, False otherwise  
*(type=boolean)*
- ts:** the timestamp that the image was taken  
*(type=float)*

#### Return Value

- None  
*(type=None)*

## 2.5 Class GPSMeasurement

A GPSMeasurement object holds information concerning one measurement taken from the aircraft's GPS. Such information includes the id of the measurement, altitude in meters, longitude, latitude, and the timestamp of when the measurement was taken.

### 2.5.1 Methods

```
__init__(self, alt, gpsId, lat, long, ts)
```

The constructor for the GPSMeasurement class.

#### Parameters

- alt:** the altitude of the measurement  
*(type=float)*
- gpsId:** the id of the GPS measurement  
*(type=int)*
- lat:** the latitude of the measurement  
*(type=float)*
- long:** the longitude of the measurement  
*(type=float)*
- ts:** the timestamp that the measurement was taken  
*(type=float)*

#### Return Value

- None  
*(type=None)*

## 2.6 Class StateMeasurement

A StateMeasurement object holds information concerning the state of the aircraft in one point in time. Such information includes the id of the measurement, roll angle of the aircraft in radians, pitch angle of the aircraft in radians, yaw angle of the aircraft in radians, and the timestamp of when the measurement was taken.

### 2.6.1 Methods

```
__init__(self, stateId, roll, pitch, yaw, ts)
```

The constructor for the StateMeasurement class.

#### Parameters

`stateId`: the id of the state measurement

*(type=int)*

`roll`: the roll angle in radians of the aircraft at the time the measurement was taken

*(type=float)*

`pitch`: the pitch angle in radians of the aircraft at the time the measurement was taken

*(type=float)*

`yaw`: The yaw angle in radians of the aircraft at the time the measurement was taken

*(type=float)*

`ts`: the timestamp that the measurement was taken

*(type=float)*

#### Return Value

`None`

*(type=None)*

## 2.7 Class ImagingInterface

The ImagingInterface object serves as the bridge between the imaging server and the imaging GUI. It connects to the imaging server given an IP address and a port. It makes certain calls to the server in order to feed the GUI what it needs such as images and measurements taken by the imaging system on the aircraft.

## 2.7.1 Methods

---

**`__init__(self, host='127.0.0.1', port='5000', numIdsStored=50, isDebug=False)`**

The constructor for the ImagingInterface class.

**Parameters**

- host:** the host of the server that the interface connects to  
(*type=String*)
- port:** the port of the server that the interface connects to  
(*type=String*)
- numIdsStored:** the number of ids that the interface keeps track of, used for getting previous images  
(*type=int*)
- isDebug:** if isDebug is true, the interface will print out status statements  
(*type=bool*)

**Return Value**

- None  
(*type=None*)

---

**`ping(self)`**

Checks to see if the interface can contact the server.

**Return Value**

- True if the interface contacted the server, False otherwise  
(*type=bool*)

---

**`debug(self, printStr)`**

If interface is in debug mode, it will print the string given, else it does nothing.

**Parameters**

- printStr:** the string that will be printed if in debug mode  
(*type=String*)

**Return Value**

- None  
(*type=None*)

**getRawImage(self, *imageId*)**

Retrieves an image with the given *imageId* from the server.

**Parameters**

*imageId*: the id of the image that is going to be returned  
*(type=int)*

**Return Value**

a tuple of the pillow Image associated with the given image id and the image id if there are any images available for processing, otherwise None  
*(type=(Image, int))*

**getNextRawImage(self, *isManual*)**

Retrieves the next available raw image from the server.

**Parameters**

*isManual*: specify whether this is a manual imaging request (True) or an autonomous one (False)  
*(type=bool)*

**Return Value**

a tuple of a pillow Image and the image id if there are any images available for processing, otherwise None  
*(type=(Image, int))*

**getPrevRawImage(self)**

Re-retrieves a raw image that was previously viewed. The interface maintains an ordered list (of up to numIdsStored) of ids that has previously been viewed and traverses the list backwards.

**Return Value**

a tuple of a pillow Image and the image id if there are any previous images to process, and the server is able to find the given id, otherwise None.  
*(type=(Image, int))*

**getImageInfo(self, *imageId*)**

Retrieves information about an image from the server given the image id.

**Parameters**

*imageId*: the id of the image of interest  
*(type=int)*

**Return Value**

an object that contains the information about the given image if it exists and connects to the server, otherwise None  
*(type=ImageInfo)*

**getCroppedImage(self, imageId)**

Retrieves a cropped image of the image from the server given the imageId.

**Parameters**

imageId: the id of the image  
(*type=int*)

**Return Value**

a tuple of a pillow Image and the image id if the image with that id is cropped, otherwise None

(*type=(Image, int)*)

**getNextCroppedImage(self)**

Retrieves the next available cropped image from the server.

**Return Value**

a tuple of a pillow Image and the image id if the image with that id is cropped, otherwise None

(*type=(Image, int)*)

**getPrevCroppedImage(self)**

Re-retrieves a cropped image that was previously viewed. The interface maintains an ordered list (of up to numIdsStored) of ids that has previously been viewed and traverses the list backwards.

**Return Value**

a pillow Image if there are any previous images to process, and the server is able to find the given id, otherwise None.

(*type=(Image, int)*)

**getCroppedImageInfo(self, imageId)**

Retrieves information about a cropped image from the server given the image id.

**Parameters**

imageId: the id of the image of interest  
(*type=int*)

**Return Value**

an object that contains the information about the given cropped image if it exists and connects to the server, otherwise None

(*type=CropInfo*)

**getAllCroppedInfo(self)**

Retrieves the information pertaining to all of the cropped images in the server.

**Return Value**

a list of CropInfo objects of all of the cropped images if it connects to the server, otherwise None

(*type=CropInfo[]*)

**imageToBytes(self, img)**

Takes an Image object and returns the bytes of the given image.

**Parameters**

img: the image to convert into bytes  
*(type=Image)*

**Return Value**

bytes of the given image  
*(type=bytes)*

**postCroppedImage(self, imageId, crop, tl, br)**

Posts a cropped image to the server.

**Parameters**

imageId: the id to the original image being cropped  
*(type=integer)*  
crop: the image file of the cropped image  
*(type=PIL Image)*  
tl: the x and y coordinate of the location of the cropped image in the top left corner relative to the original image  
*(type=integer array of length 2)*  
br: the x and y coordinate of the location of the cropped image in the bottom right corner relative to the original image  
*(type=integer array of length 2)*

**Return Value**

The response of the http request if it successfully posts, otherwise None  
*(type=Response)*

**getGPSByTs(self, ts)**

Retrieves from the server the GPS measurement that is closest to the given timestamp.

**Parameters**

ts: the timestamp of interest  
*(type=float)*

**Return Value**

GPSMeasurement object closest to the given timestamp if it connects to the server, otherwise None  
*(type=GPSMeasurement)*

**getGPSById(*self, gpsId*)**

Retrieves from the server a GPS measurement given an id.

**Parameters**

gpsId: the id of the gps measurement of interest  
*(type=int)*

**Return Value**

GPSMeasurement object of the given Id if it exists and connects to the server, otherwise None

*(type=GPSMeasurement)*

**getStateByTs(*self, ts*)**

Retrieves from the server the state measurement that is closest to the given timestamp.

**Parameters**

ts: the timestamp of interest  
*(type=float)*

**Return Value**

StateMeasurement object closest to the given timestamp if it connects to the server, otherwise None

*(type=StateMeasurement)*

**getStateById(*self, stateId*)**

Retrieves from the server a state measurement given an id.

**Parameters**

stateId: the id of the state measurement of interest  
*(type=int)*

**Return Value**

StateMeasurement object of the given Id if it exists and connects to the server, otherwise None

*(type=StateMeasurement)*



BRIGHAM YOUNG UNIVERSITY  
AUVSI CAPSTONE TEAM (TEAM 45)

---

## Geolocation Algorithm Description

---

ID	Rev.	Date	Description	Author	Checked By
IM-004	1.0	12-12-2018	Initial release	Connor Olsen	Tyler Miller
IM-004	1.0	02-20-2019	Comment / code updates	Connor Olsen	Tyler Miller

## 1 Introduction

Geolocation of targets is one of our key success measures, whose accuracy is scored by the judges for points. Accuracy is determined by distance of our location estimate from ground truth, at a max of 150ft. Anything further than 150ft will score 0 points. This geolocation algorithm is fast enough to work in real time on the server as targets are cropped.

## 2 Introduction

Given how short the algorithm is ( 150 lines), the best way to document it is through the code itself.

```
>>>
        BYU AUVSI-SUAS Capstone
        Target Geolocation Algorithm
        Connor Olsen, 2019
>>>

import numpy as np
import math as math

"""
Calculates the meters between two GPS coordinates
@type lat1: float
@param lat1: The latitude of the first GPS coordinate

@type lon1: float
@param lon1: The longitude of the first GPS coordinate

@type lat2: float
@param lat2: The latitude of the second GPS coordinate

@type lon2: float
@param lon2: The longitude of the first GPS coordinate

@rtype: array of floats(4)
@return: Distance north (meters), distande east (meters), total distance
        (meters), angle (0 deg = East)
"""

def GPSToMeters(lat1, lon1, lat2, lon2):
```

```

d2r = 0.0174532925199433
dlon = (lon2 - lon1) * d2r
dlat = (lat2 - lat1) * d2r
a = (math.sin(dlat/2.0))**2 + math.cos(lat1*d2r) * math.cos(lat2*d2r) *
    (math.sin(dlong/2.0))**2
c = 2 * math.atan2(math.sqrt(a), math.sqrt(1-a))
d = 6367 * c; #Distance between points in meters
d_meters = d * 1000
dy = lat2 - lat1
dx = math.cos(d2r * lat1) * (lon2 - lon1)
angle = math.atan2(dy, dx)
east_dis_meters = d_meters * math.cos(angle)
north_dis_meters = d_meters * math.sin(angle)
returnvalues = np.array([north_dis_meters, east_dis_meters, d_meters,
    angle])
return returnvalues

"""
Calculates GPS coordinates given a starting coordinate and meters north and
east
@type Lat: float
@param Lat: The latitude of the GPS coordinate

@type Lon: float
@param Lon: The longitude of the GPS coordinate

@type north_displacement: float
@param north_displacement: The distance north of the given coordinates (meters)

@type east_displacement: float
@param east_displacement: The distance east of the given coordinates (meters)

@rtype: array of floats(2)
@return: latitude of new target, longitude of new target
"""

def MeterstoGPS(Lat, Lon, north_displacement, east_displacement):
    # Earth[U+FFFD]s radius, sphere
    R = 6378137
    # Coordinate offsets in radians
    dLat = north_displacement/R
    dLon = east_displacement/(R*math.cos(math.pi*Lat/180))
    # OffsetPosition, decimal degrees
    lat0 = Lat + dLat * 180/math.pi

```

```
lon0 = Lon + dLon * 180/math.pi
returnvals = [lat0, lon0];
return returnvals

,
The data below will be pulled from the database:
Attitude
MAV Coordinates
Pixel Coordinates of Target
,
# For now, we will use dummy data
phi_in = 0
theta_in = 60
psi_in = 90
alpha_az = 0 # Assuming the camera is angled with the top facing out the nose
alpha_el = -math.pi/2
lat_mav = 40.2465
lon_mav = -111.6483
lat_gnd = 40.2485
lon_gnd = -111.6458
height = 16

MaxX = 2000 # Max x pixels
MaxY = 2000 # Max y pixels
,
The top left and bottom right coordinates of the cropped
photo are provided. This section finds the center of the
cropped image and translates it into
,
TopLeftX = 0
TopLeftY = 0
BottomRightX = 0
BottomRightY = 0
CenterX = BottomRightX - TopLeftX
CenterY = BottomRightY - TopLeftY
AdjustedCenterX = CenterX-(MaxX/2)
AdjustedCenterY = (-1)*(CenterY-(MaxY/2))

M = 4000
Ex = -15 #AdjustedCenterX
Ey = -1028 #-AdjustedCenterY
```

```

fov_ang = 0.872665 #field of View angle --> A6000 83* - 32* (in radians)
f = M/(2*math.tan(fov_ang/2))

l_cusp_c = 1/math.sqrt(Ex**2 + Ey**2 + f**2) * np.array([[Ex],[Ey],[f]])
"""

Convert Roll, Pitch and Yaw to radians
"""

phi = phi_in*math.pi/180
theta = theta_in*math.pi/180
psi = psi_in*math.pi/180

"""

k unit vector in the inertial frame
"""

k_i = np.array([[0],[0],[1]])

"""

Calculates distance between ground station and mav in meters to determine MAV's
relative location. Then creates the position vector [Pn Pe Pd]^T
"""

positionData = GPSToMeters(lat_gnd, lon_gnd, lat_mav, lon_mav)
P_i_mav = np.array([[positionData[0]], [positionData[1]], [-height]])

"""

Trigonometry calculated one time to decrease run time
"""

cphi = math.cos(phi)
sphi = math.sin(phi)
ctheta = math.cos(theta)
stheta = math.sin(theta)
cpsi = math.cos(psi)
spsi = math.sin(psi)
caz = math.cos(alpha_az)
saz= math.sin(alpha_az)
cel = math.cos(alpha_el) #Rreturns 6.123234e-17 instead of 0
sel = math.sin(alpha_el)

"""

Rotation from body to inertial frame
Found on page 15 of Small Unmanned Aircraft
"""

R_v2b = np.array([[ctheta*cpsi, ctheta*spsi, -stheta], \

```

```

[sphi*stheta*cpsi-cphi*spsi, sphi*stheta*spsi+cphi*cpsi, sphi*ctheta],\
[cphi*stheta*cpsi+sphi*spsi, cphi*stheta*spsi-sphi*cpsi, cphi*ctheta]])
R_b2v = np.transpose(R_v2b)
R_b2i = R_b2v

'''

R_g_to_b
Found on page 227 of Small Unmanned Aircraft
'''

R_b2g1 = np.array([[caz, saz, 0],[-saz, caz, 0],[0, 0, 1]])
R_g12g = np.array([[cel, 0, -sel],[0, 1, 0],[sel, 0, cel]])
R_b2g = np.matmul(R_g12g, R_b2g1)
R_g2b = np.transpose(R_b2g)

'''

R_c_to_g
% Found on page 227 of Small Unmanned Aircraft
'''

R_g2c = np.array([[0, 1, 0],[0, 0, 1],[1, 0, 0]])
R_c2g = np.transpose(R_g2c)

'''

% For simplicity, the three Rotation matrices are combined into one below
RbiRbgRcg = R_b_to_i * R_g_to_b * R_c_to_g;
'''

RbiRbgRcg = np.matmul(np.matmul(R_b2i, R_g2b), R_c2g)

l_cusp_i = np.matmul(RbiRbgRcg, l_cusp_c)
P_i_tar = P_i_mav + height*l_cusp_i/l_cusp_i[2]

print("Groundstation coordinates")
print(str(lat_gnd) + " " + str(lon_gnd))

print("MAV coordinates")
print(str(lat_mav) + " " + str(lon_mav))

TargetCoordinates = MeterstoGPS(lat_gnd, lon_gnd, P_i_tar[0], P_i_tar[1])
print("Target coordinates")
print(str(float(TargetCoordinates[0])) + " " +
      str(float(TargetCoordinates[1])))

```

---



BRIGHAM YOUNG UNIVERSITY  
AUVSI CAPSTONE TEAM (TEAM 45)

---

## Imaging Server API

---

ID	Rev.	Date	Description	Author	Checked By
IM-003	1.0	12-10-2018	Initial release	Tyler Miller	Brandon McBride

## Contents

<b>Contents</b>	<b>1</b>
<b>1 Introduction</b>	<b>2</b>
<b>2 Module src.dao.base_dao</b>	<b>2</b>
2.1 Class BaseDAO . . . . .	2
2.1.1 Methods . . . . .	2
2.1.2 Properties . . . . .	3
<b>3 Module src.dao.classification_dao</b>	<b>4</b>
3.1 Class ClassificationDAO . . . . .	4
3.1.1 Methods . . . . .	4
3.1.2 Properties . . . . .	6
<b>4 Module src.dao.incoming_gps_dao</b>	<b>7</b>
4.1 Class IncomingGpsDAO . . . . .	7
4.1.1 Methods . . . . .	7
4.1.2 Properties . . . . .	8
<b>5 Module src.dao.incoming_image_dao</b>	<b>9</b>
5.1 Class IncomingImageDAO . . . . .	9
5.1.1 Methods . . . . .	9
5.1.2 Properties . . . . .	10
<b>6 Module src.dao.incoming_state_dao</b>	<b>11</b>
6.1 Class IncomingStateDAO . . . . .	11
6.1.1 Methods . . . . .	11
6.1.2 Properties . . . . .	12
<b>7 Module src.dao.manual_cropped_dao</b>	<b>13</b>
7.1 Class ManualCroppedDAO . . . . .	13
7.1.1 Methods . . . . .	13
7.1.2 Properties . . . . .	15
<b>8 Package src.dao.model</b>	<b>17</b>
8.1 Modules . . . . .	17
8.2 Variables . . . . .	17
<b>9 Module src.dao.model.incoming_gps</b>	<b>18</b>
9.1 Variables . . . . .	18
9.2 Class incoming_gps . . . . .	18
9.2.1 Methods . . . . .	18
9.2.2 Properties . . . . .	18
<b>10 Module src.dao.model.incoming_image</b>	<b>20</b>
10.1 Variables . . . . .	20
10.2 Class incoming_image . . . . .	20
10.2.1 Methods . . . . .	20
10.2.2 Properties . . . . .	20

<b>11 Module src.dao.model.incoming_state</b>	<b>22</b>
11.1 Variables . . . . .	22
11.2 Class incoming_state . . . . .	22
11.2.1 Methods . . . . .	22
11.2.2 Properties . . . . .	22
<b>12 Module src.dao.model.manual_cropped</b>	<b>24</b>
12.1 Class manual_cropped . . . . .	24
12.1.1 Methods . . . . .	24
<b>13 Module src.dao.model.outgoing_autonomous</b>	<b>26</b>
13.1 Variables . . . . .	26
13.2 Class outgoing_autonomous . . . . .	26
13.2.1 Methods . . . . .	26
13.2.2 Properties . . . . .	26
<b>14 Module src.dao.model.outgoing_manual</b>	<b>29</b>
14.1 Variables . . . . .	29
14.2 Class outgoing_manual . . . . .	29
14.2.1 Methods . . . . .	29
14.2.2 Properties . . . . .	29
<b>15 Module src.dao.model.point</b>	<b>32</b>
15.1 Variables . . . . .	32
15.2 Class point . . . . .	32
15.2.1 Methods . . . . .	32
15.2.2 Properties . . . . .	33
15.2.3 Class Variables . . . . .	33
<b>16 Module src.dao.outgoing_autonomous_dao</b>	<b>34</b>
16.1 Class OutgoingAutonomousDAO . . . . .	34
16.1.1 Methods . . . . .	34
16.1.2 Properties . . . . .	36
<b>17 Module src.dao.outgoing_manual_dao</b>	<b>37</b>
17.1 Class OutgoingManualDAO . . . . .	37
17.1.1 Methods . . . . .	37
17.1.2 Properties . . . . .	39
<b>18 Module src.dao.util_dao</b>	<b>40</b>
18.1 Class UtilDAO . . . . .	40
18.1.1 Methods . . . . .	40
18.1.2 Properties . . . . .	40
<b>19 Module src.ros.ingest</b>	<b>41</b>
19.1 Functions . . . . .	41
19.2 Class RosIngest . . . . .	41
19.2.1 Methods . . . . .	41
19.2.2 Class Variables . . . . .	41
<b>Index</b>	<b>42</b>

## 1 Introduction

This document provides detailed methods of the Imaging server's database layer. Detailed explanations of the input parameters and return types of all methods are given. This document will be most useful to developers hoping to better understand the Imaging server and possibly modify it's codebase.

Note that the REST API modules (contained in the src/apis/ directory of the server) are not documented here. These modules are automatically documented by the Swagger toolchain. When the server is running, you can navigate to its homepage (localhost:5000 if running on your machine), and use the interactive documentation there to understand and try the various REST API methods.

## 2 Module src.dao.base\_dao

### 2.1 Class BaseDAO

```
object └─  
      src.dao.base_dao.BaseDAO
```

DAO with basic methods. All other DAO's are child classes of BaseDAO. Initializes and contains a postgres connection object when created.

#### 2.1.1 Methods

`__init__(self, configFilePath='..../conf/config.ini')`

Startup the DAO. Attempts to connect to the postgresql database using the settings specified in the config.ini file

Overrides: object.\_\_init\_\_

`close(self)`

Safely close the DAO's connection. It is highly recommended you call this method before finishing with a dao.

`conn(self, conn)`

`getResultingId(self, stmt, values)`

Get the first id returned from a statement. Basically this assumes you have a 'RETURNING id;' at the end of the query you are executing (insert or update)

#### Parameters

`stmt:` The sql statement string to execute

*(type=string)*

`values:` Ordered list of values to place in the statement

*(type=list)*

**executeStatements(*self, stmts*)**

Tries to execute all SQL statements in the *stmts* list. These will be performed in a single transaction. Returns nothing, so useless if you're trying to execute a series of fetches

**Parameters**

*stmts*: List of sql statements to execute  
*(type=[string])*

**basicTopSelect(*self, stmt, values*)**

Gets the first (top) row of the given select statement.

**Parameters**

*stmt*: Sql statement string to run  
*(type=string)*  
*values*: List of objects (generally int, float and string), to safely place in the sql statement.  
*(type=[object])*

**Return Value**

The first row of the select *stmt*'s result. If the statement fails or does not retrieve any records, None is returned.

*(type=[string])*

***Inherited from object***

*\_\_delattr\_\_(*self*), \_\_format\_\_(*self*), \_\_getattribute\_\_(*self*), \_\_hash\_\_(*self*), \_\_new\_\_(*self*), \_\_reduce\_\_(*self*), \_\_reduce\_ex\_\_(*self*), \_\_repr\_\_(*self*), \_\_setattr\_\_(*self*), \_\_sizeof\_\_(*self*), \_\_str\_\_(*self*), \_\_subclasshook\_\_(*self*)*

**2.1.2 Properties**

Name	Description
<i>Inherited from object</i>	
<i>__class__</i>	

## 3 Module `src.dao.classification_dao`

### 3.1 Class ClassificationDAO

```
object └  
  src.dao.base_dao.BaseDAO └  
    src.dao.classification_dao.ClassificationDAO
```

Does most of the heavy lifting for classification tables: outgoing\_autonomous and outgoing\_manual. Contains general database methods which work for both types.

#### 3.1.1 Methods

##### `__init__(self, configFilePath, outgoingTableName)`

Startup the DAO. Attempts to connect to the postgresql database using the settings specified in the config.ini file

Overrides: object.\_\_init\_\_ extit(inherited documentation)

##### `upsertClassification(self, classification)`

Upserts a classification record. If the image\_id given in the classification object already exists within the table, the corresponding record is updated. If it doesn't exist, then we insert a new record.

###### Parameters

`classification`: The outgoing\_autonomous or manual classification to upsert. Note that these objects do not require all classification properties to be successfully upserted. At a minimum it must have image\_id. ie: upsert could work if you provided a classification object with only image\_id, shape and shape\_color attributes.

*(type=outgoing\_manual)*

###### Return Value

The resulting table id (Note: not image\_id) of the classification row if successfully upserted, otherwise -1

*(type=int)*

**addClassification(self, classification)**

Adds the specified classification information to one of the outgoing tables

**Parameters**

**classification:** The classifications to add to the database  
*(type=outgoing-autonomous or outgoing-manual)*

**Return Value**

Id of classification if inserted, otherwise -1  
*(type=int)*

**getClassificationByUID(self, id)**

Attempts to get the classification with the specified universal-identifier

**Parameters**

**id:** The id of the image to try and retrieve  
*(type=int)*

**getClassification(self, id)**

Gets a classification by the TABLE ID. This is opposed to getClassificationByUID, which retrieves a row based off of the unique image\_id. This is mostly used internally, and is not used by any of the public REST API methods.

**Parameters**

**id:** The table id of the classification to retrieve.  
*(type=int)*

**Return Value**

String list of values retrieved from the database. Child classes will properly place these values in model objects. If the given id doesn't exist, None is returned.

*(type=[string])*

**getAll(self)**

Get all the images currently in this table

**Return Value**

A cursor to the query result for the specified classification type. This allows children classes to place the results in their desired object type.

*(type=cursor)*

**updateClassificationByUID(*self, id, updateClass*)**

Builds an update string based on the available key-value pairs in the given classification object if successful, returns an classification object of the entire row that was updated

**Parameters**

**id:** The image\_id of the classification to update

*(type=int)*

**updateClass:** Information to attempt to update for the classification with the provided image\_id

*(type=outgoing\_autonomous or outgoing\_manual)*

**Return Value**

The classification of the now updated image\_id if successful.

Otherwise None

*(type=outgoing\_autonomous or outgoing\_manual)*

**getAllDistinct(*self, modelGenerator, whereClause=None*)**

Get all the unique classifications in the classification queue Submitted or not.

*Inherited from src.dao.base\_dao.BaseDAO(Section 2.1)*

basicTopSelect(), close(), conn(), executeStatements(), getResultingId()

*Inherited from object*

\_\_delattr\_\_(), \_\_format\_\_(), \_\_getattribute\_\_(), \_\_hash\_\_(), \_\_new\_\_(), \_\_reduce\_\_(), \_\_reduce\_ex\_\_(),  
 \_\_repr\_\_(), \_\_setattr\_\_(), \_\_sizeof\_\_(), \_\_str\_\_(), \_\_subclasshook\_\_()

### 3.1.2 Properties

Name	Description
<i>Inherited from object</i>	
__class__	

## 4 Module *src.dao.incoming\_gps\_dao*

### 4.1 Class IncomingGpsDAO

```
object └  
  src.dao.base_dao.BaseDAO └  
    src.dao.incoming_gps_dao.IncomingGpsDAO
```

Handles interaction with recorded GPS measurements. Ros\_ ingest interacts with this DAO directly. On the REST side, most of its functionality is accessed through the /gps endpoint

#### 4.1.1 Methods

##### `__init__(self, configFilePath)`

Startup the DAO. Attempts to connect to the postgresql database using the settings specified in the config.ini file

Overrides: object.\_\_init\_\_ extit(inherited documentation)

##### `addGps(self, incomingGps)`

Adds the specified image to the incoming\_image table

###### Parameters

*incomingGps*: The gps measurement to add to the database  
        (*type=incoming\_gps*)

###### Return Value

    Id of gps measurement if successfully inserted, otherwise -1  
        (*type=int*)

**getGpsById(*self, id*)**

Get a gps measurement from the database by its id. This will likely only be used internally, if at all.

**Parameters**

**id:** The unique id of the gps measurement to retrieve  
*(type=int)*

**Return Value**

An incoming\_gps object with all recorded gps information if the measurement with the given id exists, otherwise None.

*(type=incoming\_gps)*

**getGpsByClosestTS(*self, ts*)**

Get the gps row that has a time\_stamp closest to the one specified. This will likely be the method most used by geolocation and autonomous localization methods.

**Parameters**

**ts:** UTC Unix Epoch timestamp as a float. The closest gps measurement to this timestamp will be returned  
*(type=float)*

**Return Value**

An incoming\_gps object will all the recorded gps information for the measurement closest to the provided timestamp. Note that if the provided timestamp is lower than all timestamp measurements or if the gps table is empty, None will be returned.

*(type=incoming\_gps)*

**Inherited from src.dao.base\_dao.BaseDAO(Section 2.1)**

basicTopSelect(), close(), conn(), executeStatements(), getResultingId()

**Inherited from object**

\_\_delattr\_\_(), \_\_format\_\_(), \_\_getattribute\_\_(), \_\_hash\_\_(), \_\_new\_\_(), \_\_reduce\_\_(), \_\_reduce\_ex\_\_(),  
 \_\_repr\_\_(), \_\_setattr\_\_(), \_\_sizeof\_\_(), \_\_str\_\_(), \_\_subclasshook\_\_()

**4.1.2 Properties**

Name	Description
<i>Inherited from object</i>	

## 5 Module `src.dao.incoming_image_dao`

### 5.1 Class IncomingImageDAO

```
object └  
  src.dao.base_dao.BaseDAO └  
    src.dao.incoming_image_dao.IncomingImageDAO
```

Handles interaction with raw images captured by the plane. Ros\_ ingest interacts with this DAO directly. On the REST side, most of its functionality is accessed through the /image/raw endpoint and the raw\_image\_handler module

#### 5.1.1 Methods

##### `__init__(self, configFilePath)`

Startup the DAO. Attempts to connect to the postgresql database using the settings specified in the config.ini file

Overrides: object.\_\_init\_\_ (inherited documentation)

##### `addImage(self, incomingImage)`

Adds the specified image to the incoming\_image table

###### Parameters

    incomingImage: The image to add to the database  
                  (type=incoming\_image)

###### Return Value

    Id of image if successfully inserted, otherwise -1  
    (type=int)

**getImage(self, id)**

Attempts to get the image with the specified id

**Parameters**

**id:** The id of the image to try and retrieve  
*(type=int)*

**Return Value**

An incoming\_image with the info for that image if successfully found,  
 otherwise None  
*(type=incoming\_image)*

**getNextImage(self, manual)**

Attempts to get the next raw image not handled by the specified mode  
 (manual or autonomous)

**Parameters**

**manual:** Whether to try and get the next image in manual's queue  
 (True) or the autonomous queue (False)  
*(type=bool)*

**Return Value**

An incoming\_image with the info for that image if successfully found,  
 otherwise None  
*(type=incoming\_image)*

**Inherited from src.dao.base\_dao.BaseDAO(Section 2.1)**

basicTopSelect(), close(), conn(), executeStatements(), getResultingId()

**Inherited from object**

`__delattr__(self), __format__(self), __getattribute__(self), __hash__(self), __new__(cls), __reduce__(self), __reduce_ex__(self),  
 __repr__(self), __setattr__(self), __sizeof__(self), __str__(self), __subclasshook__(self)`

### 5.1.2 Properties

Name	Description
<i>Inherited from object</i>	
<code>__class__</code>	

## 6 Module `src.dao.incoming_state_dao`

### 6.1 Class IncomingStateDAO

```
object └  
  src.dao.base_dao.BaseDAO └  
    src.dao.incoming_state_dao.IncomingStateDAO
```

Handles interaction with recorded state measurements. Ros\_ ingest interacts with this DAO directly. On the REST side, most of its functionality is accessed through the /state endpoint

#### 6.1.1 Methods

##### `__init__(self, configFilePath)`

Startup the DAO. Attempts to connect to the postgresql database using the settings specified in the config.ini file

Overrides: object.`__init__` extit(inherited documentation)

##### `addState(self, incomingState)`

Adds the specified image to the incoming\_image table

###### Parameters

`incomingState`: The state measurements to add to the database  
        (*type=incoming\_state*)

###### Return Value

    Id of state measurements if successfully inserted, otherwise -1  
        (*type=int*)

**getStateById(*self, id*)**

Get a state measurement from the database by its id. This will likely only be used internally, if at all.

**Parameters**

**id:** The unique id of the state measurement to retrieve  
*(type=int)*

**Return Value**

An incoming\_state object with all recorded state information if the measurement with the given id exists, otherwise None.

*(type=incoming\_state)*

**getStateByClosestTS(*self, ts*)**

Get the state row that has a time\_stamp closest to the one specified. This will likely be the method most used by geolocation and autonomous localization methods.

**Parameters**

**ts:** UTC Unix Epoch timestamp as a float. The closest state measurement to this timestamp will be returned  
*(type=float)*

**Return Value**

An incoming\_state object will all the recorded state information for the measurement closest to the provided timestamp. Note that if the provided timestamp is lower than all timestamp measurements or if the state table is empty, None will be returned.

*(type=incoming\_state)*

**Inherited from src.dao.base\_dao.BaseDAO(Section 2.1)**

basicTopSelect(), close(), conn(), executeStatements(), getResultingId()

**Inherited from object**

\_\_delattr\_\_(), \_\_format\_\_(), \_\_getattribute\_\_(), \_\_hash\_\_(), \_\_new\_\_(), \_\_reduce\_\_(), \_\_reduce\_ex\_\_(),  
 \_\_repr\_\_(), \_\_setattr\_\_(), \_\_sizeof\_\_(), \_\_str\_\_(), \_\_subclasshook\_\_()

### 6.1.2 Properties

Name	Description
<i>Inherited from object</i> __class__	

## 7 Module *src.dao.manual\_cropped\_dao*

### 7.1 Class ManualCroppedDAO

```
object └  
  src.dao.base_dao.BaseDAO └  
    src.dao.manual_cropped_dao.ManualCroppedDAO
```

#### 7.1.1 Methods

##### ***\_\_init\_\_(self, configFilePath)***

Startup the DAO. Attempts to connect to the postgresql database using the settings specified in the config.ini file

Overrides: object.*\_\_init\_\_* extit(inherited documentation)

##### ***upsertCropped(self, manualCropped)***

Upserts a cropped image record. Will only insert or update values persent in the parameter

###### Parameters

**manualCropped:** manual\_cropped object to update or insert. At a minimum must contain an image\_id  
        *(type=manual\_cropped)*

###### Return Value

    Internal table id of the record inserted/updated if successful.

    Otherwise -1

*(type=int)*

**`addImage(self, manualCropped)`**

Adds the manually cropped image to the manual\_cropped table

**Parameters**

`manualCropped`: manual\_cropped image to insert. Should have  
 image\_id, time\_stamp, cropped\_path, and tapped  
 $(type=manual\_cropped)$

**Return Value**

Internal table id of the manual\_cropped entry if successfully inserted,  
 otherwise -1  
 $(type=int)$

**`getImageByUID(self, id)`**

Attempts to get the image with the specified universal-identifier

**Parameters**

`id`: The id of the image to try and retrieve  
 $(type=int)$

**Return Value**

A manual\_cropped image with the info for that image if successfully found, otherwise None  
 $(type=manual\_cropped)$

**`getImage(self, id)`**

Attempts to get the image with the specified manual\_cropped table id. NOTE: the different between getImageByUID. getImageByUID selects on the image\_id which is a universal id for an image shared across the incoming\_image, manual\_cropped and outgoing\_manual tables

**Parameters**

`id`:  $(type=int)$

**Return Value**

manual\_cropped instance that was retrieved. If no image with that id exists, None  
 $(type=manual\_cropped)$

**getNextImage(*self*)**

Get the next un-tapped cropped image for classification. This will retrieve the oldest cropped image where 'tapped'=FALSE.

**Return Value**

The next available manual\_cropped image if one is available, otherwise None

*(type=manual\_cropped)*

**getAll(*self*)**

Get all the cropped image currently in the table

**Return Value**

List of all cropped images in the manual\_cropped table. If the table is empty, an empty list

*(type=[outgoing\_manual])*

**updateImageByUID(*self, id, updateContent*)**

Update the image with the specified image\_id.

**Parameters**

**id:** Image\_id of the cropped information to update  
*(type=int)*

**updateContent:** Dictionary/JSON of attributes to update  
*(type={ object})*

**Return Value**

manual\_cropped instance showing the current state of the now-updated row in the table. If the update fails, None

*(type=manual\_cropped)*

**Inherited from *src.dao.base\_dao.BaseDAO*(Section 2.1)**

`basicTopSelect()`, `close()`, `conn()`, `executeStatements()`, `getResultingId()`

**Inherited from *object***

`__delattr__()`, `__format__()`, `__getattribute__()`, `__hash__()`, `__new__()`, `__reduce__()`, `__reduce_ex__()`,  
`__repr__()`, `__setattr__()`, `__sizeof__()`, `__str__()`, `__subclasshook__()`

### 7.1.2 Properties

Name	Description
<i>Inherited from object</i> __class__	

## 8 Package *src.dao.model*

### 8.1 Modules

- **incoming\_gps** (*Section 9, p. 18*)
- **incoming\_image** (*Section 10, p. 20*)
- **incoming\_state** (*Section 11, p. 22*)
- **manual\_cropped** (*Section 12, p. 24*)
- **outgoing\_autonomous** (*Section 13, p. 26*)
- **outgoing\_manual** (*Section 14, p. 29*)
- **point** (*Section 15, p. 32*)

### 8.2 Variables

Name	Description
<code>-package--</code>	<b>Value:</b> None

## 9 Module src.dao.model.incoming\_gps

### 9.1 Variables

Name	Description
_package_	<b>Value:</b> None

### 9.2 Class incoming\_gps

Model class for the Gps table. Properties and helper methods for gps measurements.

#### 9.2.1 Methods

`__init__(self, tableValues=None)`

`insertValues(self)`

Get the gps measurement as an object list. The properties are ordered as they would be for a normal table insert

**Return Value**

Ordered object list - time\_stamp, lat, lon, alt

(*type*=[object])

`toDict(self)`

Return properties contained in this measurement as a dictionary

**Return Value**

Object dictionary of gps measurement properties

(*type*={ object })

`__str__(self)`

Debug convenience method to get this instance as a string

#### 9.2.2 Properties

Name	Description
id	Table id for this measurement. Empty when inserting.

*continued on next page*

Name	Description
time_stamp	UTC Unix epoch timestamp as float.
lat	Measurement latitude as a float
lon	Measurement longitude as a float
alt	Measurement altitude as a float

## 10 Module src.dao.model.incoming\_image

### 10.1 Variables

Name	Description
_package_	<b>Value:</b> None

### 10.2 Class incoming\_image

Model class for the Raw Image table. Properties and helper methods for raw images from the camera.

#### 10.2.1 Methods

`__init__(self, tableValues=None)`

`insertValues(self)`

Get the raw image as an object list. The properties are ordered as they would be for a normal table insert

**Return Value**

Ordered object list - time\_stamp, image\_path, manual\_tap, autonomous\_tap

*(type=[object])*

`toDict(self)`

Return properties contained in this model as a dictionary

**Return Value**

Object dictionary of raw image properties

*(type={object})*

`__str__(self)`

Debug convenience method to get this instance as a string

#### 10.2.2 Properties

Name	Description
image_id	Table id for this image. This image_id is used throughout The other tables as a unique identifier back to various states of the image.
time_stamp	UTC Unix epoch timestamp as float.
image_path	Path to where the image is saved on the server filesystem
manual_tap	Boolean indicating whether this image has been 'tapped' (aka seen) by the manual imaging client
autonomous_tap	Boolean indicating whether this image has been 'tapped' (aka seen) by the autonomous imaging client

## 11 Module src.dao.model.incoming\_state

### 11.1 Variables

Name	Description
_package_	<b>Value:</b> None

### 11.2 Class incoming\_state

Model class for the State table. Properties and helper methods for state measurements.

#### 11.2.1 Methods

`__init__(self, tableValues=None)`

`insertValues(self)`

Get the gps measurement as an object list. The properties are ordered as they would be for a normal table insert

**Return Value**

Ordered object list - time\_stamp, roll, pitch, yaw  
`(type=[object])`

`toDict(self)`

Return properties contained in this measurement as a dictionary

**Return Value**

Object dictionary of state measurement properties  
`(type={ object })`

`__str__(self)`

Debug convenience method to get this instance as a string

#### 11.2.2 Properties

Name	Description
id	Table id for this measurement. Empty when inserting a new measurement.

*continued on next page*

Name	Description
time_stamp	UTC Unix epoch timestamp as float.
roll	Measurement roll as a float
pitch	Measurement pitch as a float
yaw	Measurement yaw as a float

## 12 Module src.dao.model.manual\_cropped

### 12.1 Class manual\_cropped

Model class for the manual\_cropped table. Properties and helper methods for images cropped by the manual client

#### 12.1.1 Methods

**\_init\_\_(self, tableValues=None, json=None)**

Accepts various formats to instantiate this model object

##### Parameters

tableValues: List of table values, in table column order

(*type*=*[object]*)

json: Json dictionary of table values. Used by the REST API when receiving data

(*type*=*{ object}*)

**id(self, id)**

**image\_id(self, image\_id)**

**time\_stamp(self, time\_stamp)**

**cropped\_path(self, cropped\_path)**

**crop\_coordinate\_tl(self, crop\_coordinate\_tl)**

**crop\_coordinate\_br(self, crop\_coordinate\_br)**

**tapped(self, tapped)**

**allProps(self)**

**toDict(*self*, *exclude*=None)**

Return attributes contained in this model as a dictionary

**Parameters**

*exclude*: Attribute names to exclude from the generated result  
*(type=(string))*

**Return Value**

String dictionary of cropped image properties  
*(type={string})*

**toJsonResponse(*self*, *exclude*=None)**

Produce a dictionary of this cropped instance. This is very similar to the `toDict` method, but adds a few values to the json to separate crop coordinates into x and y

**Parameters**

*exclude*: Attribute names to exclude from the generated result  
*(type=(string))*

**Return Value**

Dictionary of attributes stored in this instance, not including those attributes specified in `exclude`.  
*(type={object})*

**insertValues(*self*)**

Get the cropped image as an object list. The properties are ordered as they would be for a barebones table insert. (In many cases crop coordinates are provided for the initial insert, so this method isn't used)

**Return Value**

Ordered object list - `image_id`, `time_stamp`, `cropped_path`, `tapped`  
*(type=[object])*

## 13 Module src.dao.model.outgoing\_autonomous

### 13.1 Variables

Name	Description
_package_	<b>Value:</b> None

### 13.2 Class outgoing\_autonomous

Model class for the autonomous classification 'outgoing\_autonomous' table. This model is very similar to the outgoing\_manual model class.

#### 13.2.1 Methods

##### `__init__(self, tableValues=None, json=None)`

Accepts various formats to instantiate this model object

###### Parameters

`tableValues`: List of table values, in table column order

*(type=[object])*

`json`: Json dictionary of table values. Used by the REST API when receiving data

*(type={ object})*

##### `allProps(self)`

##### `toDict(self, exclude=None)`

Return attributes contained in this model as a dictionary

###### Parameters

`exclude`: Attribute names to exclude from the generated result

*(type=(string))*

###### Return Value

String dictionary of classification properties

*(type={ string})*

#### 13.2.2 Properties

Name	Description
id	Table id. Internal to the dao, not exposed by the REST API
image_id	Unique image.id, publicly exposed by the API and used to access information on the image in various states (raw, cropped, and classified)
type	Type of classification. AUVSI currently specifies three possible types: 'standard', 'off_axis' or 'emergent'. Type must equal one of these to be successfully inserted or modified in the table
latitude	Geolocation latitude of the object
longitude	Geolocation longitude of the object
orientation	Orientation of the character/object. AUVSI currently specifies 8 possible orientations: 'N', 'NE', 'E', 'SE', 'S', 'SW', 'W' or 'NW'. Orientation must equal one of these to be successfully inserted or modified in the table.
shape	Shape of the object for standar/off-axis types. AUVSI currently specifies 13 possible shapes: 'circle', 'semicircle', 'quarter_circle', 'triangle', 'square', 'rectangle', 'trapezoid', 'pentagon', 'hexagon', 'heptagon', 'octagon', 'star' or 'cross'. Shape must equal one of these to be successfully inserted or modified in the table.
background_color	Background color of the object for standard/off-axis types. AUVSI currently specifies 10 possible colors: 'white', 'black', 'gray', 'red', 'blue', 'green', 'yellow', 'purple', 'brown' or 'orange'. Background_color must equal one of these to be successfully inserted or modified in the table
alphanumeric	Alphanumeric within the target for standard/off-axis target types. At present AUVSI specifies that any uppercase alpha character or number may be within a target. Through in practice they have historcall only done alpha characters. Checking that this property is given/contains valid values is left to the user.

*continued on next page*

Name	Description
alphanumeric_color	Color of the alphanumeric for a standard/off-axis type. Color specs are the same as background_color. Alphanumeric_color must be equal to one of the specified colors to be successfully inserted or modified in the table.
description	Description of the emergent object.
submitted	Boolean to indicate whether the classification has been submitted to the judges yet

## 14 Module src.dao.model.outgoing\_manual

### 14.1 Variables

Name	Description
_package_	<b>Value:</b> None

### 14.2 Class outgoing\_manual

Model class for the manual classification 'outgoing\_manual' table. This model is very similar to the outgoing\_autonomous model class.

#### 14.2.1 Methods

##### `__init__(self, tableValues=None, json=None)`

Accepts various formats to instantiate this model object

###### Parameters

`tableValues`: List of table values, in table column order

*(type=[object])*

`json`: Json dictionary of table values. Used by the REST API when receiving data

*(type={ object})*

##### `allProps(self)`

##### `toDict(self, exclude=None)`

Return attributes contained in this model as a dictionary

###### Parameters

`exclude`: Attribute names to exclude from the generated result

*(type=(string))*

###### Return Value

String dictionary of classification properties

*(type={ string})*

#### 14.2.2 Properties

Name	Description
id	Table id. Internal to the dao, not exposed by the REST API
image_id	Unique image.id, publicly exposed by the API and used to access information on the image in various states (raw, cropped, and classified)
type	Type of classification. AUVSI currently specifies three possible types: 'standard', 'off_axis' or 'emergent'. Type must equal one of these to be successfully inserted or modified in the table
latitude	Geolocation latitude of the object
longitude	Geolocation longitude of the object
orientation	Orientation of the character/object. AUVSI currently specifies 8 possible orientations: 'N', 'NE', 'E', 'SE', 'S', 'SW', 'W' or 'NW'. Orientation must equal one of these to be successfully inserted or modified in the table.
shape	Shape of the object for standar/off-axis types. AUVSI currently specifies 13 possible shapes: 'circle', 'semicircle', 'quarter_circle', 'triangle', 'square', 'rectangle', 'trapezoid', 'pentagon', 'hexagon', 'heptagon', 'octagon', 'star' or 'cross'. Shape must equal one of these to be successfully inserted or modified in the table.
background_color	Background color of the object for standard/off-axis types. AUVSI currently specifies 10 possible colors: 'white', 'black', 'gray', 'red', 'blue', 'green', 'yellow', 'purple', 'brown' or 'orange'. Background_color must equal one of these to be successfully inserted or modified in the table
alphanumeric	Alphanumeric within the target for standard/off-axis target types. At present AUVSI specifies that any uppercase alpha character or number may be within a target. Through in practice they have historcall only done alpha characters. Checking that this column is given/contains valid values is left to the user.

*continued on next page*

Name	Description
alphanumeric_color	Color of the alphanumeric for a standard/off-axis type. Color specs are the same as background_color. Alphanumeric_color must be equal to one of the specified colors to be successfully inserted or modified in the table.
description	Description of the emergent object.
submitted	Boolean to indicate whether the classification has been submitted to the judges yet

## 15 Module src.dao.model.point

### 15.1 Variables

Name	Description
_package_	<b>Value:</b> 'src.dao.model'

### 15.2 Class point

Represents a point datatype from postgres. Used by manual\_cropped model for crop\_coordinates

#### 15.2.1 Methods

##### `__init__(self, ptStr=None, x=None, y=None)`

Provides various ways to initialize different point types

###### Parameters

`ptStr`: String of a integer point, should look something like:

“(45,56)”

(*type=string*)

`x`: Integer for the x component of the point

(*type=int*)

`y`: Integer for the y component of the point

(*type=int*)

##### `toSql(self)`

Generate a string that can be successfully inserted as a point into postgres.

Requires both x and y attributes to be present.

###### Return Value

String representing the point. Formatted: (x,y). If x or y is not present, None.

(*type=string*)

**toDict(*self*)**

Return attributes contained in this model as a dictionary

**Return Value**

String dictionary of point properties. If x or y is not present, None

(*type*=*{int}*)

**\_\_str\_\_(*self*)**

Debug convenience method to get this instance as a string

### 15.2.2 Properties

Name	Description
x	X component of the point
y	Y component of the point

### 15.2.3 Class Variables

Name	Description
INT_REGEX	Value: '[^\\d]*(\\d)[^\\d]*'

## 16 Module *src.dao.outgoing\_autonomous\_dao*

### 16.1 Class OutgoingAutonomousDAO

```
object └  
  src.dao.base_dao.BaseDAO └  
    src.dao.classification_dao.ClassificationDAO └  
      src.dao.outgoing_autonomous_dao.OutgoingAutono
```

Outgoing\_autonomous wrapper for the ClassificationDAO. Most of the core functionality here happens in the ClassificationDAO

#### 16.1.1 Methods

**`__init__(self, configFilePath)`**

Startup the DAO. Attempts to connect to the postgresql database using the settings specified in the config.ini file

Overrides: object.\_\_init\_\_ extit(inherited documentation)

**`checkedReturn(self, rawResponse)`**

**`getClassificationByUID(self, id)`**

See classification\_dao docs. Here we're just making sure we cast the final object to the proper outgoing classification model type

#### Parameters

`id`: The id of the image to try and retrieve

Overrides: src.dao.classification\_dao.ClassificationDAO.getClassificationByUID

### `getAll(self)`

See classification\_dao docs. Here we're just making sure we cast the final object to the proper outgoing classification model type

#### Return Value

A cursor to the query result for the specified classification type. This allows children classes to place the results in their desired object type.

*(type=cursor)*

Overrides: src.dao.classification\_dao.ClassificationDAO.getAll

### `getClassification(self, id)`

See classification\_dao docs. Here we're just making sure we cast the final object to the proper outgoing classification model type

#### Parameters

`id`: The table id of the classification to retrieve.

#### Return Value

String list of values retrieved from the database. Child classes will properly place these values in model objects. If the given id doesn't exist, None is returned.

*(type=[string])*

Overrides: src.dao.classification\_dao.ClassificationDAO.getClassification

### `updateClassificationByUID(self, id, updateClass)`

See classification\_dao docs. Here we're just making sure we cast the final object to the proper outgoing classification model type. We're also properly setting up the initial model of stuff to update before passing it to super

#### Parameters

`id`: The image\_id of the classification to update

`updateClass`: Information to attempt to update for the classification with the provided image\_id

#### Return Value

The classification of the now updated image\_id if successful.

Otherwise None

*(type=outgoing\_autonomous or outgoing\_manual)*

Overrides:

src.dao.classification\_dao.ClassificationDAO.updateClassificationByUID

**getAllDistinct(*self*)**

Get all the unique classifications in the classification queue Submitted or not.

Overrides: src.dao.classification\_dao.ClassificationDAO.getAllDistinct  
 extit(inherited documentation)

**getAllDistinctPending(*self*)**

Get images grouped by distinct targets pending submission (ei: submitted = false)

**newModelFromRow(*self, row*)**

A reflective function for the classification dao. Pass self up to the super ClassificationDAO. It calls this method to create the proper model object in its response. Not uber elegant, but presently used by getAllDistinct.

**Parameters**

**row:** List of ordered string values to be placed within an outgoing\_autonomous object  
*(type=[string])*

*Inherited from src.dao.classification\_dao.ClassificationDAO(Section 3.1)*

addClassification(), upsertClassification()

*Inherited from src.dao.base\_dao.BaseDAO(Section 2.1)*

basicTopSelect(), close(), conn(), executeStatements(), getResultingId()

*Inherited from object*

\_\_delattr\_\_( ), \_\_format\_\_( ), \_\_getattribute\_\_( ), \_\_hash\_\_( ), \_\_new\_\_( ), \_\_reduce\_\_( ), \_\_reduce\_ex\_\_( ),  
 \_\_repr\_\_( ), \_\_setattr\_\_( ), \_\_sizeof\_\_( ), \_\_str\_\_( ), \_\_subclasshook\_\_( )

### 16.1.2 Properties

Name	Description
<i>Inherited from object</i>	
__class__	

## 17 Module src.dao.outgoing\_manual\_dao

### 17.1 Class OutgoingManualDAO

```
object └  
  src.dao.base_dao.BaseDAO └  
    src.dao.classification_dao.ClassificationDAO └  
      src.dao.outgoing_manual_dao.OutgoingManualDAO
```

Outgoing\_manual wrapper for the ClassificationDAO. Most of the core functionality here happens in the ClassificationDAO

#### 17.1.1 Methods

**`__init__(self, configFilePath)`**

Startup the DAO. Attempts to connect to the postgresql database using the settings specified in the config.ini file

Overrides: object.\_\_init\_\_ extit(inherited documentation)

**`checkedReturn(self, rawResponse)`**

**`getClassificationByUID(self, id)`**

See classification\_dao docs. Here we're just making sure we cast the final object to the proper outgoing classification model type

#### Parameters

`id`: The id of the image to try and retrieve

Overrides: src.dao.classification\_dao.ClassificationDAO.getClassificationByUID

**getAll(*self*)**

See classification\_dao docs. Here we're just making sure we cast the final object to the proper outgoing classification model type

**Return Value**

A cursor to the query result for the specified classification type. This allows children classes to place the results in their desired object type.

*(type=cursor)*

Overrides: src.dao.classification\_dao.ClassificationDAO.getAll

**getClassification(*self, id*)**

See classification\_dao docs. Here we're just making sure we cast the final object to the proper outgoing classification model type

**Parameters**

**id:** The table id of the classification to retrieve.

**Return Value**

String list of values retrieved from the database. Child classes will properly place these values in model objects. If the given id doesn't exist, None is returned.

*(type=[string])*

Overrides: src.dao.classification\_dao.ClassificationDAO.getClassification

**updateClassificationByUID(*self, id, updateClass*)**

See classification\_dao docs. Here we're just making sure we cast the final object to the proper outgoing classification model type. We're also properly setting up the initial model of stuff to update before passing it to super

**Parameters**

**id:** The image\_id of the classification to update

**updateClass:** Information to attempt to update for the classification with the provided image\_id

**Return Value**

The classification of the now updated image\_id if successful.

Otherwise None

*(type=outgoing\_autonomous or outgoing\_manual)*

Overrides:

src.dao.classification\_dao.ClassificationDAO.updateClassificationByUID

**`getAllDistinct(self)`**

Get all the unique classifications in the classification queue Submitted or not.

Overrides: `src.dao.classification_dao.ClassificationDAO.getAllDistinct`  
 extit(inherited documentation)

**`getAllDistinctPending(self)`**

Get images grouped by distinct targets pending submission (ei: submitted = false)

**`newModelFromRow(self, row)`**

Kinda a reflective function for the classification dao. Pass self up to the super ClassificationDAO, and it calls this method to create the proper model object in its response.

Not uber elegant, only used by `getAllDistinct` atm.

*Inherited from src.dao.classification\_dao.ClassificationDAO(Section 3.1)*

`addClassification()`, `upsertClassification()`

*Inherited from src.dao.base\_dao.BaseDAO(Section 2.1)*

`basicTopSelect()`, `close()`, `conn()`, `executeStatements()`, `getResultingId()`

*Inherited from object*

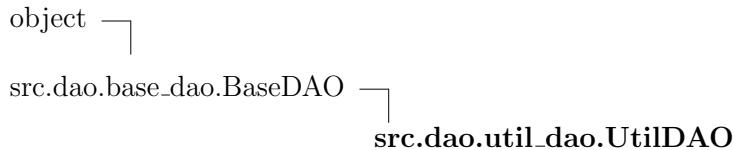
`__delattr__(self)`, `__format__(self, format_spec)`, `__getattribute__(self, name)`, `__hash__(self)`, `__new__(cls, *args, **kwargs)`, `__reduce__(self)`, `__reduce_ex__(self)`,  
`__repr__(self)`, `__setattr__(self, name, value)`, `__sizeof__(self)`, `__str__(self)`, `__subclasshook__(self)`

### 17.1.2 Properties

Name	Description
<i>Inherited from object</i>	
<code>__class__</code>	

## 18 Module src.dao.util\_dao

### 18.1 Class UtilDAO



Holds utility methods to help manage the database

#### 18.1.1 Methods

**`__init__(self, configFilePath)`**

Startup the DAO. Attempts to connect to the postgresql database using the settings specified in the config.ini file

Overrides: object.\_\_init\_\_ extit(inherited documentation)

**`resetManualDB(self)`**

Resets the database to an initial form as if a rosbag was just read in

**`resetAutonomousDB(self)`**

Resets the database to an initial form as if a rosbag was just read in

*Inherited from src.dao.base\_dao.BaseDAO(Section 2.1)*

basicTopSelect(), close(), conn(), executeStatements(), getResultingId()

*Inherited from object*

\_\_delattr\_\_(), \_\_format\_\_(), \_\_getattribute\_\_(), \_\_hash\_\_(), \_\_new\_\_(), \_\_reduce\_\_(), \_\_reduce\_ex\_\_(),  
 \_\_repr\_\_(), \_\_setattr\_\_(), \_\_sizeof\_\_(), \_\_str\_\_(), \_\_subclasshook\_\_()

#### 18.1.2 Properties

Name	Description
<i>Inherited from object</i>	
__class__	

## 19 Module src.ros\_ingest

### 19.1 Functions

**main()**

### 19.2 Class RosIngestor

This script is the bridge between ROS and the rest of the imaging system. It's only objective is listen to the ros network and save relevant information to the server's database. Subscribes to the raw image, state and gps ros topics

#### 19.2.1 Methods

**\_\_init\_\_(self)**

**gpsCallback(self, msg)**

Ros subscriber callback. Subscribes to the inertial\_sense GPS msg. Get the lla values from the GPS message and then pass them to the DAO so they can be inserted into the database

**stateCallback(self, msg)**

Ros subscriber callback. Subscribes to the /state rosplane topic. Passes the roll, pitch and yaw angle to be saved by the DAO.

**imgCallback(self, msg)**

Ros subscriber callback. Subscribes to the cameras image topic. Saves the image file, and passes the corresponding filename and TS to the DAO so that it can be inserted into the database

#### 19.2.2 Class Variables

Name	Description
STATE_SAVE_EVERY	<b>Value:</b> 10

## Index

src (*package*)  
  src.dao (*package*)  
    src.dao.base\_dao (*module*), 2–3  
    src.dao.classification\_dao (*module*), 4–6  
    src.dao.incoming\_gps\_dao (*module*), 7–8  
    src.dao.incoming\_image\_dao (*module*), 9–10  
    src.dao.incoming\_state\_dao (*module*), 11–12  
    src.dao.manual\_cropped\_dao (*module*), 13–16  
    src.dao.model (*package*), 17  
    src.dao.outgoing\_autonomous\_dao (*module*), 34–36  
    src.dao.outgoing\_manual\_dao (*module*), 37–39  
    src.dao.util\_dao (*module*), 40  
  src.ros\_ingest (*module*), 41  
    src.ros\_ingest.main (*function*), 41  
    src.ros\_ingest.RosIngeste (*class*), 41



BRIGHAM YOUNG UNIVERSITY  
AUVSI CAPSTONE TEAM (TEAM 45)

---

## Imaging Test Procedures

---

ID	Rev.	Date	Description	Author	Checked By
IM-006	1.0	12-11-2018	Initial release	Tyler Miller	Jake Johnson
IM-006	1.1	02-21-2019	Add geolocation test procedures	Connor Olsen	Tyler Miller

## 1 Introduction

In order to verify that the imaging subsystem independently meets its key success measures a series of tests and procedures were developed. Requirements for the subsystem can be seen in IM-007. However, some of the subsystem's requirements cannot be fully measured until it is integrated with the full system and testing in flight.

## 2 Continuous Integration

A prerequisite to meeting our key success measures, is a reliable, bug-free codebase. Imaging undertook a large task this year with our server-client model requiring us to rewrite our entire codebase. This rewrite presented an opportunity to build a proper test framework in parallel with our server-client code. A series of unit and integration tests were developed for each component of the server and client. These tests are extremely beneficial because they ensure proper functionality as changes are made to the code. Tests are run automatically by our github repository on every commit and test results are reported back to the team. This allows us to pinpoint breaking changes and respond to them quickly.

The CI tests start with a series of unit tests to examine server-database functionality. A unit test was created for every Database Access Object (DAO) method used by the server to interact with the database. These methods perform basic database operations, such as insert, select, update, and delete. The unit tests perform these operations through the DAO methods, and then confirm that the database reflects the expected results of such operation. These methods also inherently test constraints placed within the database itself (ie: primary key and unique constraints).

Next, integration test are performed. The purpose of these tests are to verify the client rest library (described in IM-001) interacts with the server as expected. The client rest library presents top-level methods useful to the client gui to interact with the imaging server. Thus integration tests confirm that the client rest library and the api endpoint handlers on the server function properly.

Regression tests, which are added as bugs are found and patched, naturally fall within one of the existing unit or integration tests and are placed there. Between these unit and integration tests, we can confidently say the server performs as expected. Having a verified codebase allows us to reliably perform tests on the system in the future as we integrate with other subsystems.

### 3 Bandwidth Test Procedure

One of our most important success measures, as described in the imaging requirements matrix (IM-007) is streaming images at a high enough rate, so we're able to perform classifications in an actionable (near real time) manner. A high stream rate will also guarantee that we don't miss any targets as the plane performs its search pattern since higher stream rates will allow more image overlap.

The bandwidth test itself is fairly simple to perform and can be done on the ground.

1. With the plane's onboard computer running, connect it to the network using the ubiquiti bullet and litebeam, as it would be connected during an actual flight.
2. On a computer connected to the router over ethernet, confirm you are on the proper ros network.
3. Start the ros handler code with: `rosrun imaging_ros_handler ros_handler.py`
4. On the plane, confirm the camera is connected to the onboard computer and start the camera driver with: `rosrun a6000_ros_node a6000_ros`
5. On the server machine (connected to the router), monitor the image stream rate using: `rostopic hz`, this will print out the average number of messages/second every few seconds.
6. Walk the plane away from the base station, tracking distance and stream rate every 10-20 feet.

With this method we saw stream rates between 1.5 - 2.5Hz. It was also observed that the main rate limiter was the camera itself. Since the camera captures and compresses the image before sending it to the plane's onboard computer, it seems to consistently take 1s for capture and 1 additional second for it to be successfully sent to the computer (presumably this time is used to compresses the image before it is transmitted). Given the camera's high resolution, wide field of view, and the plane's slow speed; this rate should be adequate to capture the entire competition field.

### 4 Geolocation Test Procedure

Without the Camera being fully ready to take pictures in a flight test, there was need to simulate an environment where the accuracy of the target geolocation algorithm could be verified. Using our own equipment, we were able to create a controlled environment to verify the accuracy within 20 feet.

**Equipment used:**

- Sony A6000 Camera
- Tripod
- Target
- iPhone (to calculate angles and take in GPS coordinates)

### **Procedure**

We took the target to the alleyway between the Clyde and Engineering Buildings and looked for a good spot to mount the camera. Without access to the roof, we ended up using the next best option, which was the fourth floor skybridge that connects the two buildings. We measured the tripod at a -30 degree angle, (corresponding to the plane pitching 60 degrees, since the camera is always 90 degree downward from the nose) and facing directly east (corresponding to the plane rolling 90 degrees)

We measured the height of the camera relative to the target to be approximately 16 meters, and took the GPS coordinates of both the camera and the target. The resulting image is shown in figure 1 below



*Figure 1: AUVSI geolocation test result*

When plugged into the algorithm, the results were extremely and obviously incorrect. This error helped us isolate and resolve a minor bug that was affecting the pitch. With the code working correctly, the algorithm was able to determine the coordinates of the target in the image with only it's own coordinates, it's states (roll, pitch, yaw) and the pixel coordinates of the target (shown in the image).

The result was accurate within 20 feet, which is already a major success. Once we have more images taken from our plane to continue testing with, we believe that the accuracy will improve, as we will be using more reliable GPS data, and not google maps.