Brigham Young University
AUVSI Capstone Team (Team 45)

Subsystem Engineering Artifacts

Table of Contents

Imaging Test Procedures

Brigham Young University

AUVSI Capstone Team (Team 45)

Capstone Project Contract

Andrew Torgesen, 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

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^{th} to 15^{th}, 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^{th} overall while the 2018 team was a Capstone team and placed 9^{th} 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	jacobcjohnson13@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

Development Stage	**Expected Completion Date**	**Design Artifacts Required for Approval**	**Budget**
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**	**Excellent (A)**	**Good (B)**	**Fair (C)**	**Lower Acceptable**	**Ideal**	**Upper Acceptable**
Obstacles Hit (#)	0	1	3	5	0	0	5
Average Waypoint Proximity (ft)*	5	20	25	30	0	0	100
Characteristics Identified (%)**	80	40	30	20	20	100	100
Airdrop Accuracy (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)

UAS Subsystem Interface Definition

ID	Rev.	Date	Description	Author	Checked By
SS-001	0.1	10-25-2018	initial draft	Andrew Torgesen	Jake Johnson & John Akagi
SS-001	0.2	10-30-2018	adjusted wording	Andrew Torgesen	Kameron Eves
SS-001	1.0	10-30-2018	adjusted diagram	Andrew Torgesen	Brady Moon
SS-001	1.1	11-05-2018	added introduction and fixed typos	Andrew Torgesen	Brady Moon

1 Introduction

At its heart, the AUVSI competition is a systems engineering competition, testing how well a team can bring together a complex amalgamation of software and hardware components to accomplish sophisticated tasks in autonomy and aviation. While no key success measure directly measures this integration, all of the key success measures are acheived through adequate system integration. Thus, as part of the Concept Development process for the UAS, proper interface protocols must be defined so that inter-component testing can commence as soon as possible. Upon identifying the most critical subsystem interfaces, tests may be designed to evaluate the effectiveness of our chosen means of communicating between subsystems.

2 Subsystem Interfaces

Figure 1 gives a top-level description of the major hardware and software subsystems, as well as how they interface in the fully-functioning UAS. Table 1 lists descriptions of the functions of each software component listed in the figure.

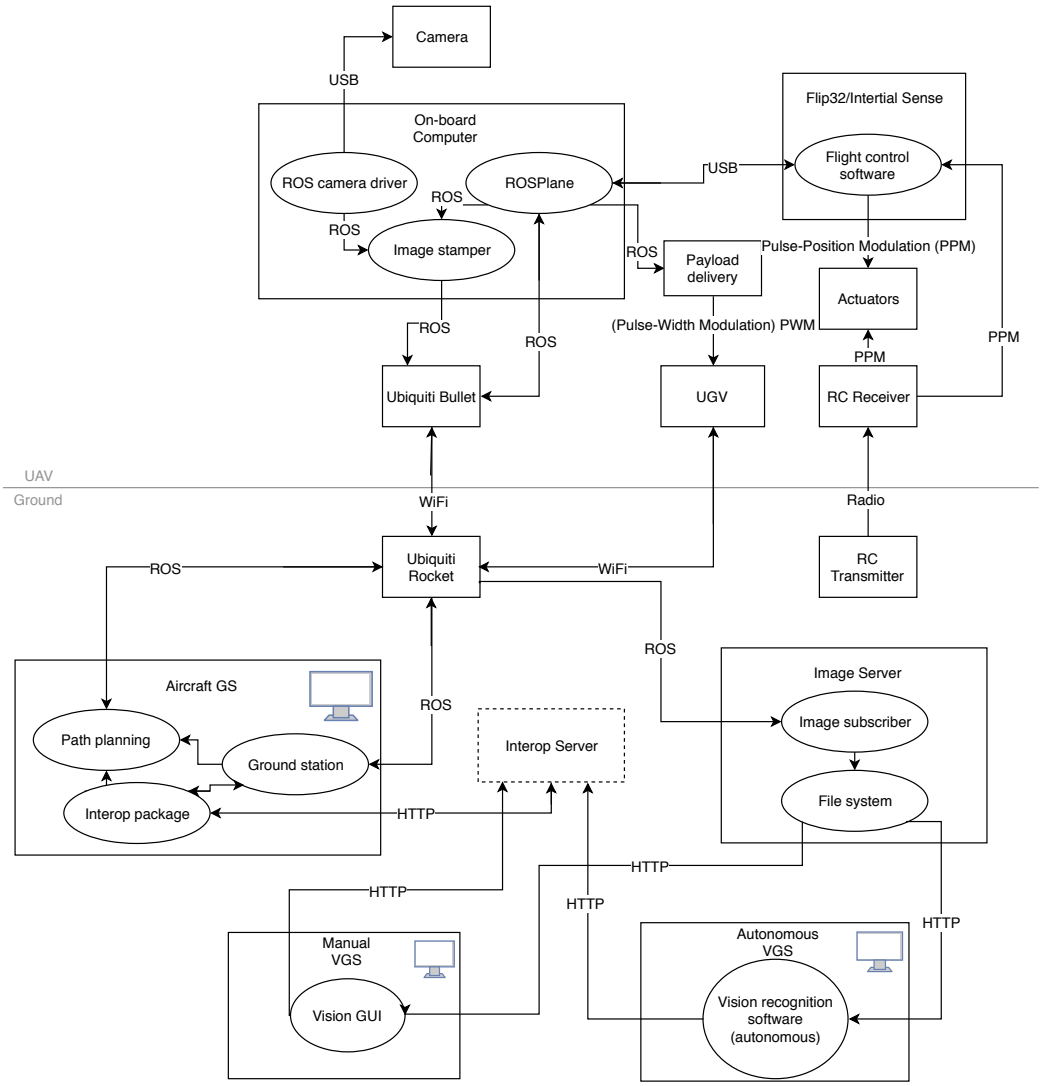


Figure 1: System-wide interface diagram for the UAS. Hardware is denoted by a box, and software is denoted by an oval.

Table 1: Descriptions of the functions of the software components listed in Figure 1.

Software Component	**Description**
ROS camera driver	Reads the serial input from the camera and streams it as ROS messages so other ROS programs have access to the camera images in real time.
ROSPlane	Top-level autopilot. Takes a set of waypoints and converts them into low-level commands to be interpreted by the flight control software. Also constructs a state vector containing all of the dynamic states of the UAS.
Image stamper	Takes streamed camera images and stamps them with time and UAS state data. This facilitates subsequent geolocation of objects found in each image.
Flight control software	Converts low-level autopilot commands into actuation commands and reads in sensor data. Consists of: • ROSFlight: handles autopilot commands, reads in airspeed and barometer data • Inertial Sense: reads in GPS and inertial sensor data
Path planning	Given the details of the competition (including obstacle and flight area data), plans a series of waypoints for the UAS.
ground station	Allows for the visualization of the UAS and provides an interface for sending waypoint, loiter, and return-to-home commands.
Interop package	Communicates with the judges' interop server, and serves up competition details over the ROS network. Also reports UAS data back to the judges' server.
Image subscriber	Captures streamed camera images from the ROS network.
File system	Stores images from Image subscriber on the computer's file system for direct HTTP access by ground station computers.
Vision GUI	Provides an interface for the manual classification of targets in images, as well as reporting the classification data to the judges' server.
Vision recognition software (autonomous)	Runs computer vision software that autonomously classifies targets in images and reports the results to the judges' server.

3 Conclusion

As can be seen from Figure 1, both radio and WiFi will be used to facilitate connection between the subsystems on the ground and in the air. The Ubiquiti data link allows for communication between the ground and the aircraft over a WiFi network. A 2.4 GHz radio link (independent) between the radio transmitter and receiver allows for manual control and arming/disarming of the aircraft.

The Robot Operating System (ROS) is what facilitates the majority of inter-component communication over the WiFi network. ROS is a Linux middle-ware and development protocol for creating modular programs for robotics. ROS allows for real-time communication between machines running individual nodes, or executables, over a WiFi network. In our system, all subsystems communicating via ROS either are or will be developed as ROS nodes to be run on a machine with Linux installed. For more information about ROS nodes and how they communicate over a network, see http://www.ros.org/.

Brigham Young University

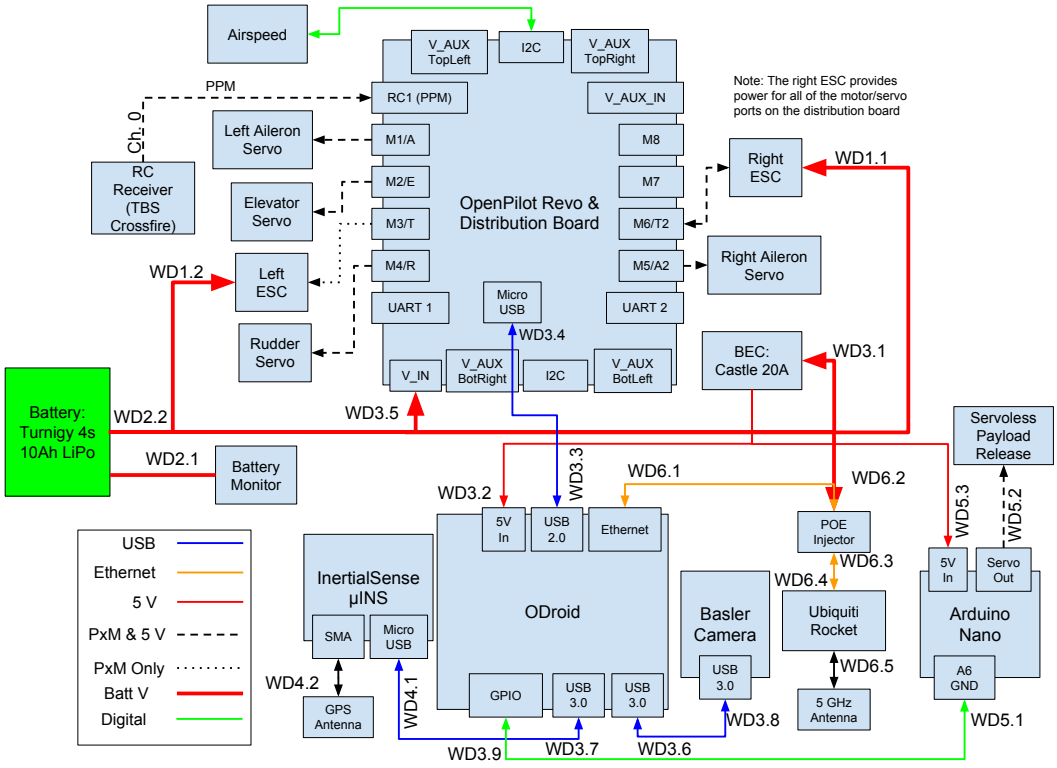
AUVSI Capstone Team (Team 45)

Wiring Diagram

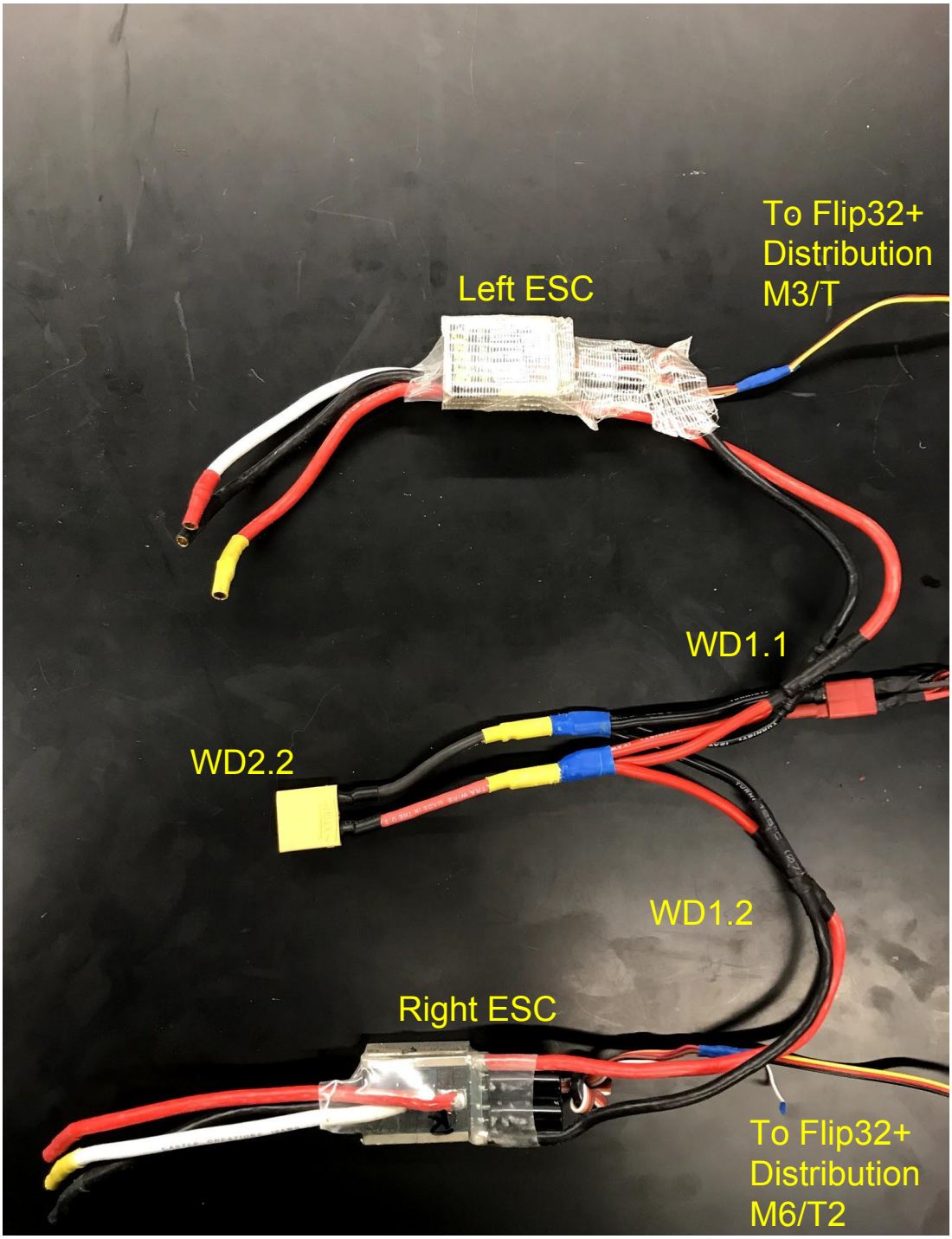
ID	Rev.	Date	Description	Author	Checked By
AF-005	0.1	12-10-2018	Created	Jacob Willis	Andrew Torgesen

Introduction

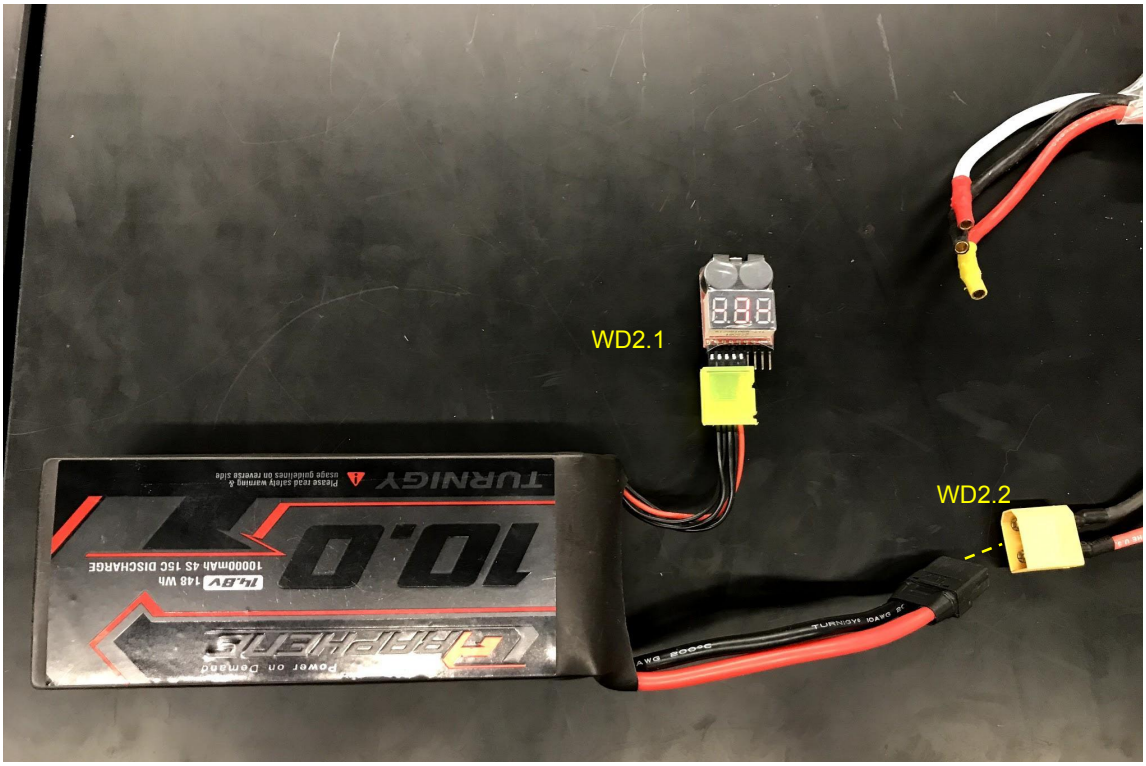
This artifact depicts the wiring connections between electrical components within the airframe. Each connection lable corresponds to a page in the diagram and a connection on that page, so WD5.1 refers to connection 1 on page 5. Connections that appear on more than one page are given the page number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



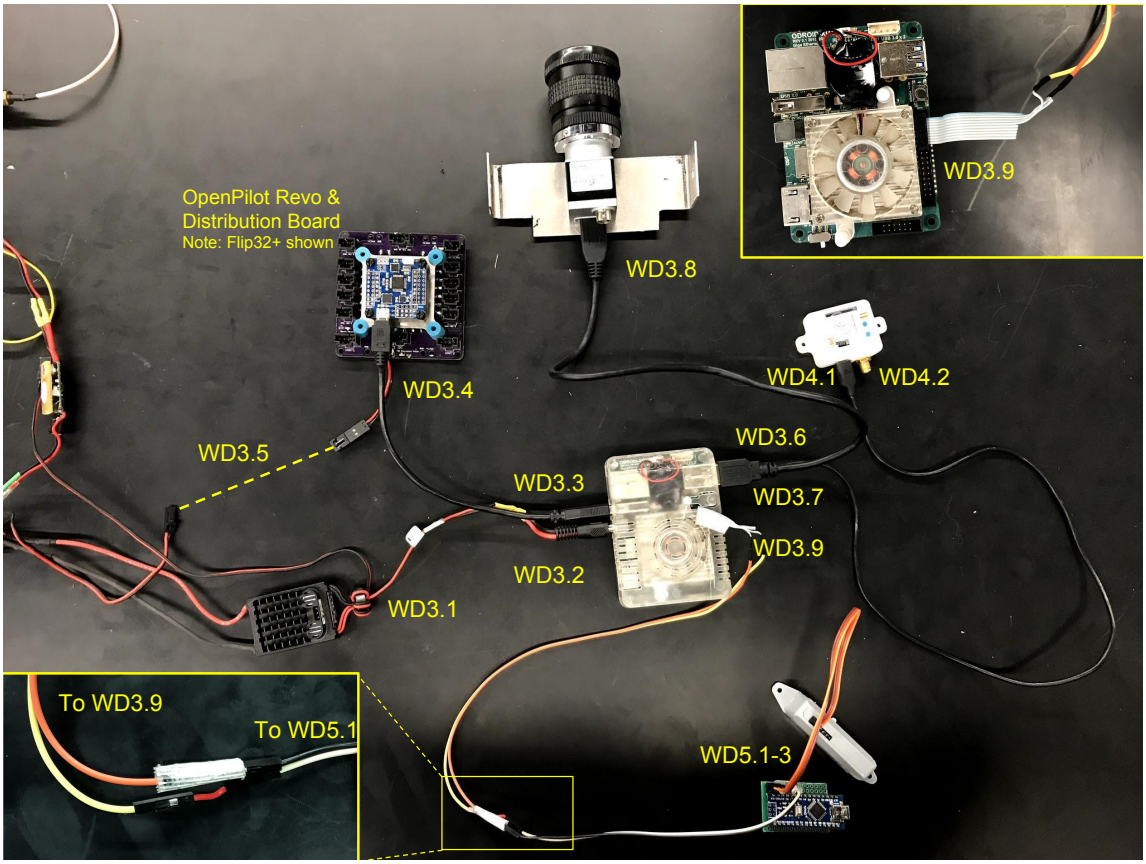
Top level logical diagram.The power harness consists of the Batt V connections, and the BEC 20A converter.


To Flip32+
Distribution
M3/T
Left ESC
WD1.1
WD2.2
WD1.2
Right ESC
To Flip32+
Distribution
M6/T2

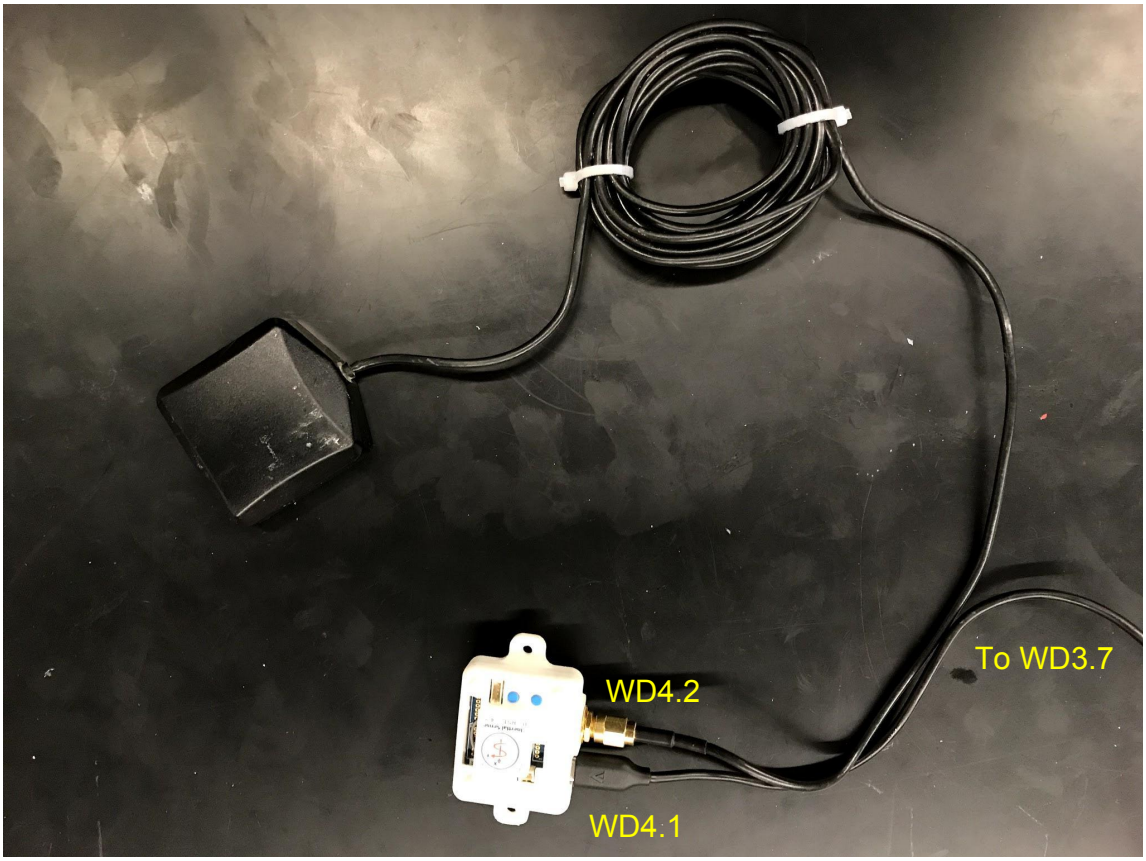
WD1



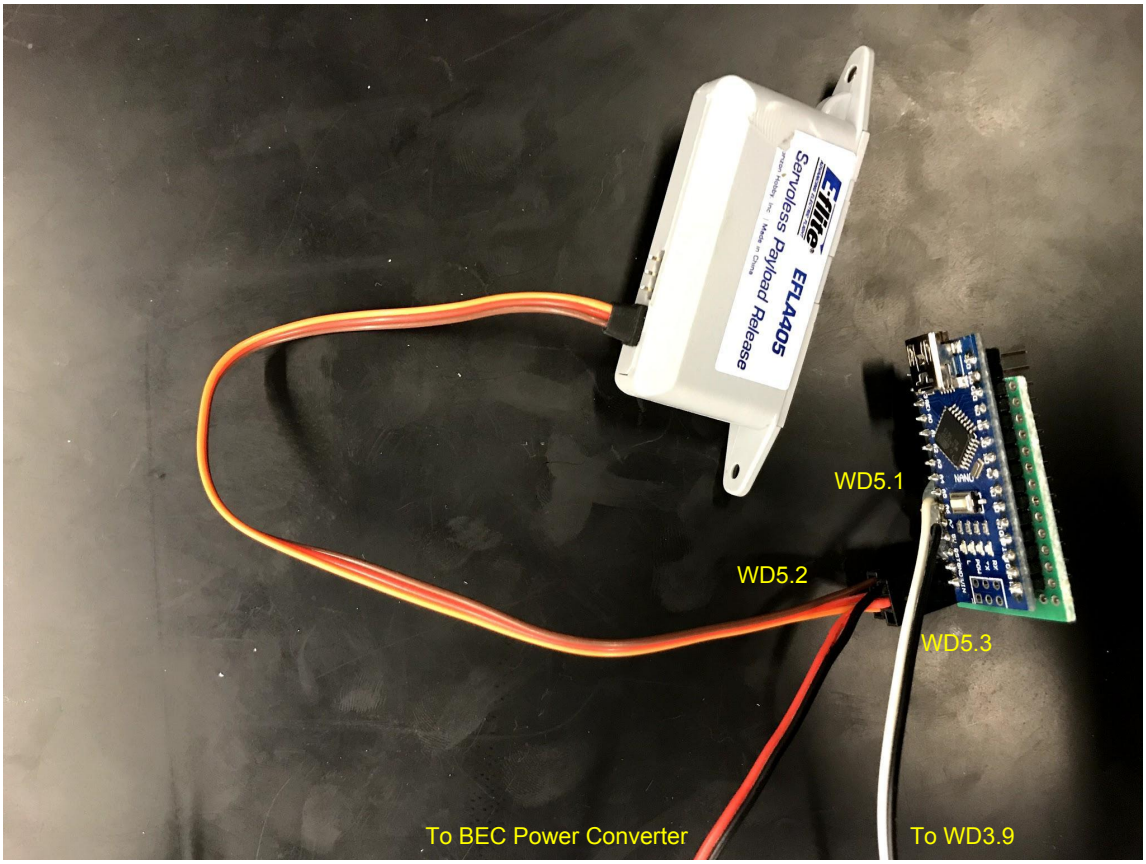
WD2


OpenPilot Revo &
Distribution Board
Note: Flip32+ shown
WD3.8
WD3.9
WD4.1
WD4.2
WD3.4
WD3.5
WD3.6
WD3.3
WD3.7
WD3.9
WD3.2
WD3.1
To WD3.9
To WD5.1
WD5.1-3

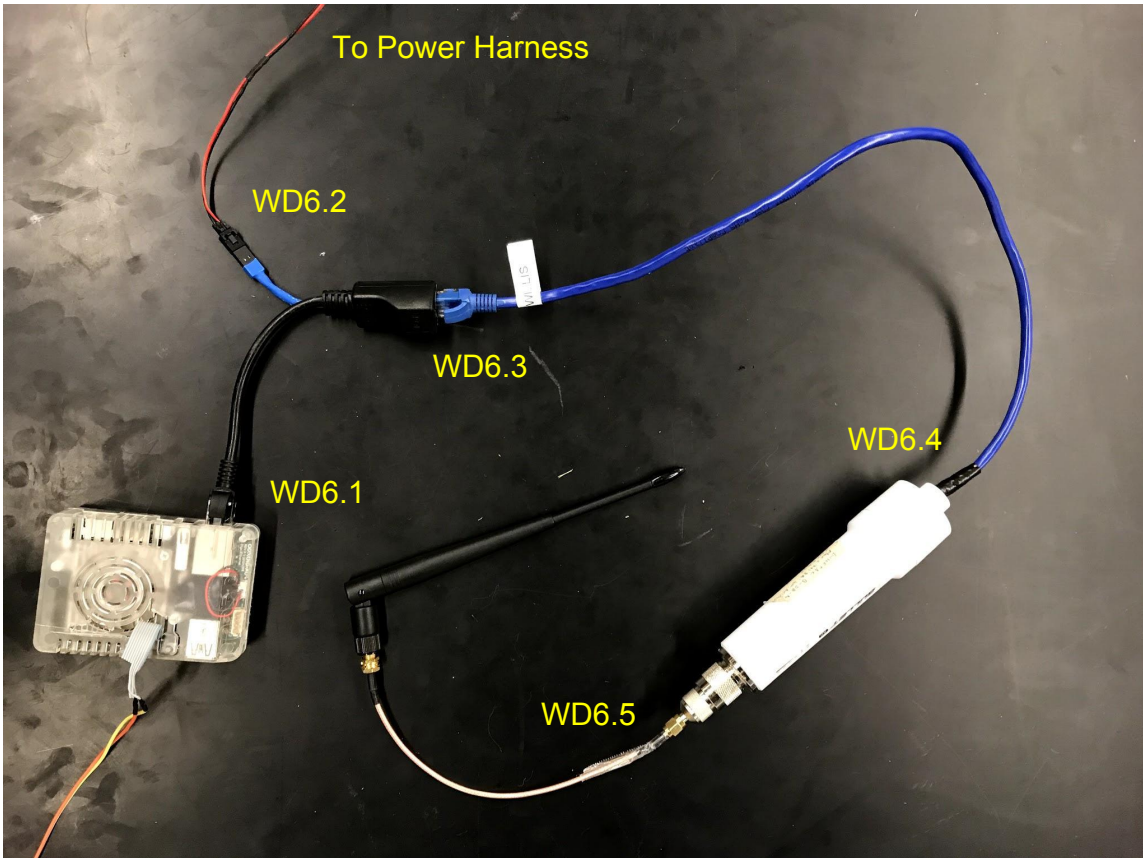
WD3



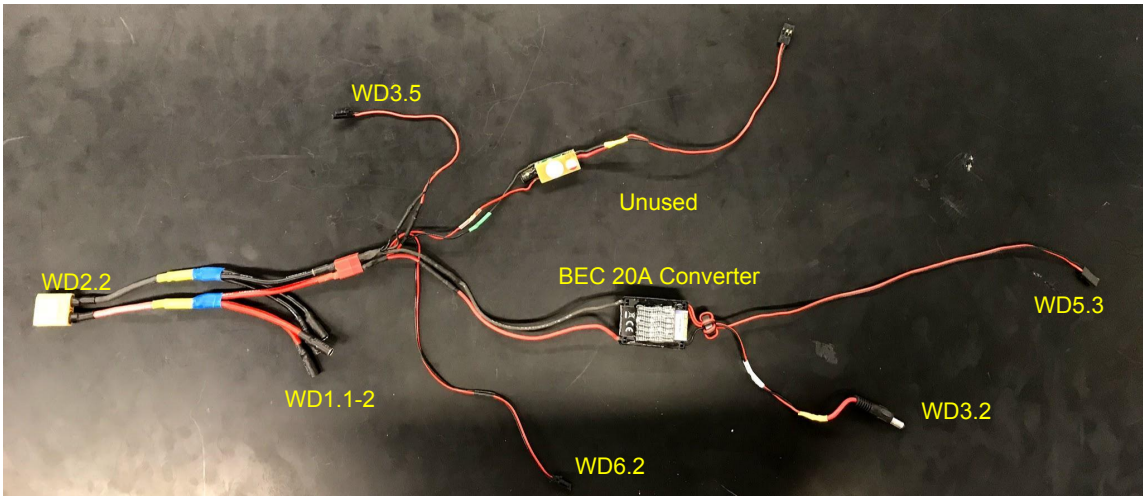
WD4


Servoless Payload Release
EFL405
WD5.1
WD5.2
WD5.3
To BEC Power Converter
To WD3.9

WD5



WD6



Power harness

Brigham Young University

AUVSI 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	0.3	2-5-2019	Added Take-off/Landing Tracker	Kameron Eves	Brandon McBride

*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 **Auto (min)**	Takeoffs/ Landings **Auto**	**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 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 batter 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.

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.

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.

2-6-19	Utah County Airfield	17 **4**	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	31 **10**	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.

Statistics

Total Flight Time: 2 Hour 7 Minutes

Manual Flight Time: 1 Hour 52 Minutes*

Autonomous Flight Time: 15 Minutes

Percent of Autonomous Flight: 11.8%

Manual Takeoffs: 19*

Manual Landings: 13*

Autonomous Takeoffs: 0

Autonomous Landings: 0

Percent of Flights Ending in Crash: 31.6%

Flights Sense Last Crash: 13

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

Brigham Young University

AUVSI Capstone Team (Team 45)

Field Flight Checklist v1.0

ID	Rev.	Date	Description	Author	Checked By
PF-001	0.1	11-03-2018	Wrote checklist based on google sheet and research	Andrew Torgesen	Brandon McBride
PF-001	0.2	01-07-2019	Updated checklist based on team feedback	Andrew Torgesen	Tyler Miller
PF-001	1.0	02-04-2019	Removed redundant checks and added RC override info	Andrew Torgesen	Kameron Eves

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
- ☐ Connect battery
- ☐ 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 ($> 10\ mW$, 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)

Bill of Materials

ID	Rev.	Date	Description	Author	Checked By
BM-001	0.1	02-19-2019	SS Engineering Draft	Tyler Critchfield	Brandon McBride

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 current Nimbus Pro 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 irrelevent (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	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	Items 4, 11, 16, 17, 19	1	99	Hard Kernel
16	Camera	Items 15, 28	1	241	Basler
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	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	N/A	Amazon
34	UGV GPS	Item 32, 37	1	N/A	N/A
35	Parachute housing	Item 36	1	10	Custom
36	30” 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

Brigham Young University

AUVSI Capstone Team (Team 45)

Airframe Subsystem Description

ID	Rev.	Date	Description	Author	Checked By
AF-008	0.1	02-19-19	Initial Draft for Subsystem Engineering	Tyler Critchfield	Ryan Anderson

1 Introduction

This artifact describes the final design and modification of the Nimbus Pro airframe that was selected during the Concept Development stage. Images of the constructed airframe are included as well as a list of modifications we made to the plane. The advantages of the current concept in achieving our key success measures are described.

2 Design Description

The Nimbus Pro airframe 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.

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

A detailed procedure for constructing the plane is beyond the scope of this artifact. It suffices to say that the procedure follows basic principles and steps of foam aircraft construction, as follows:

- Glue fuselage sections together
- Attach proper servos and control horns to each control surface
- Attach motors and ESCs to wings

- Attach tail pieces
- Install all electronic hardware/wiring
- Ensure connectors and joints are properly secured and strengthened if necessary
- Attach/connect wings

The center of gravity (CG) was carefully placed to optimize performance using an aerodynamics model, as explained in AF-011. The following modifications were made to the airframe based on our application and integrated hardware:

- Holes were made in the main compartment hatch for the R/C and Ubiquiti antennas. There is dedicated space further back in the plane for these components, but placing components there moved the CG back too far for an acceptable static margin. Care was also taken to space the Ubiquiti antenna far from the GPS antenna to avoid signal interference.
- The GPS slot on top was not used and taped over. The GPS was instead placed inside the nose of the plane to move the CG forward.
- The servos we selected were larger than the airframe was designed for. Because of this, some foam and plastic was removed from servo locations using razor blades, a hot metal tool, and wire cutters to allow them to fit.
- A foam wedge was inserted underneath the tail to increase the tail incidence angle (see further details below).
- Because the motor power wires are so thick, we bypassed the electrical connector that came with the airframe. Small holes were drilled into the wing connector pieces to allow for the large wires to extend from the ESCs in the wings to the fuselage.
- In general, most components were placed as far forward as was reasonable to move our CG forward and increase our static margin.

In Concept Development, we considered adding wing extensions to increase total span. This would help the plane fly slower by increasing lift. Flying at a lower velocity would then help us achieve higher performance in our key success measures (specifically obstacles hit, waypoint proximity, characteristics identified, and accuracy of payload drop). In order to decide if wing extensions would be worth, the aerodynamic benefits of extensions were modeled and compared to the relative cost of manufacture (See AF-012). In summary, it was decided that any extra benefit to extensions would not be worth the time and effort required to design, implement and test these extensions.

One unanticipated modification we've made to the aircraft is the placement of a foam wedge underneath the tail of the plane. Interestingly, the factory design has no tail

incidence. Though we predicted problems, we decided to see if the plane would fly without this modification. It did fly, but it had a consistent tendency to want to pitch forward, making it difficult to take off and fly R/C. The plane was modeled and an incidence angle of 7.5° determined the optimal condition (see AF-009 and AF-011). After installing the tail wedge, this problem was averted. The plane is now stable and flies very near the design speed we had originally planned for.

The tail wedge (see Fig. 2) is made out of extruded polystyrene, measured and cut manually in order to give the required incidence angle for the tail. Several notches were carved into the top for it to securely fit with the tail connector pieces. Fiber tape was then wrapped around it to ensure it would not come loose during flight. The design drawing for the wedge is included in AF-010.

Figure 2: A foam tail wedge installed underneath the tail of the plane to increase its incidence angle and improve stability.

3 Testing

As described further in Artifact AF-013, we have done extensive flight testing to ensure our airframe performs as expected. In the past couple of months, we have had several successful flight tests with manual RC control and multiple successful flights while controlling with autopilot. While flying, it is very stable and flies at an average of 15 m/s. These outcomes show that our airframe works and is very capable to integrate with the other subsystems. Not only have we flown it with some autopilot control, we have also shown it can hold the imaging subsystem camera and the UGV subsystem. Both have

demonstrated their subsystems can work with the plane while stationary, and we have yet to test them in flight. From these results, however, we are very confident our airframe will integrate well with these subsystems while in flight.

4 Conclusion

Our airframe is fully assembled (with slight modifications) and has proven to consistently fly reliably. All of our key success measures depend on the airframe flying well - and many of them can be improved if the airframe flies with a low velocity. Our design and modifications ensure the airframe does fly slowly, which will help minimize obstacles hit, waypoint proximity, and error when dropping the UGV payload. It also improves image quality to identify more characteristics. The plane has flown successfully under R/C and autopilot control, the payload drop system has functioned successfully in flight, and the imaging subsystems have performed well inside the plane while stationary. We fully expect all three other subsystems to successfully integrate with the airframe during flight.

Brigham Young University

AUVSI Capstone Team (Team 45)

Failure Modes and Effects Analysis

ID	Rev.	Date	Description	Author	Checked By
AF-007	0.1	02-19-19	Initial creation	Kameron Eves	Tyler Critchfield

1 Introduction

To mitigate risk of failure within the competition, a Failure Modes and Effects Analysis (FMEA) was performed. Many deficiencies were found and were then corrected to an acceptable level.

2 Analysis

Component	Functional Purpose	Failure Mode*	Failure Effect	Failure Cause	Current Situation				Assigned Action	Improved Situation			
					S	L	D	RPN		S	L	D	RPN
RC Reciver	Communicate Manual Commands from the RC Transmittor to F4	Hardware Failure*	Mission Failure Aircraft Loiters	Poorly Connected Electrical Joint	8	1	7	56		8	1	7	56
		Transmits incorrect data	Crash	Internal Code	9	1	10	90	Extensive testing prior to use**	9	1	10	90
		Loss of Connection	Mission Failure Aircraft Loiters	Interference	8	4	9	288	FFCL*** range test	8	4	3	96
RC Transmittor	Communicate Commands from the RC Pilot to the RC Reciver	Hardware Failure	Mission Failure Aircraft Loiters	Poorly Connected Electrical Joint	8	2	7	112	FFCL	8	2	3	48
		Transmits incorrect data	Crash	Settings Incorrect	9	2	6	108	FFCL	9	1	4	36
		Loss of Connection	Mission Failure Aircraft Loiters	Settings Incorrect	8	6	8	384	FFCL	8	4	3	96
				Interference	8	4	9	288	FFCL	8	4	3	96
				Transmittor Battery Dead	8	6	3	144	FFCL	8	4	2	64
WIFI antenna	Allow communitation with grounstation over ROS network	Hardware Failure	Mission Failure Manual Landing	Poorly Connected Electrical Joint	6	1	7	42		6	1	7	42
		Loss of Connection	Mission Failure Manual Landing	Antenna Incorrectly Pointed	6	7	3	126	Assign someone to point antenna	6	3	3	54
Odroid	Run ROS, generate high level commands, process images, & estimate state	Hardware failure	Mission Failure Manual Landing	Poorly Connected Electrical Joint	6	2	7	84		6	2	7	84
			Crash	Poorly Connected Electrical Joint	9	1	7	63	Extensive testing prior to use	9	1	3	27
F4 Flight Computer & Mount	Turn high level (Odroid & RC) commands into low level servo commands	Software Failure	Crash	Internal Code	9	3	6	162	Extensive testing prior to use	9	3	3	81
		Hardware Failure	Crash	Poorly Connected Electrical Joint	9	3	7	189	Extensive testing prior to use	9	3	3	81
Airspeed Sensor	Measure Va	Software Failure	Flight Less Smooth	Internal Code	4	1	10	40		4	1	10	40
		Inaccurate Readings	Flight Less Smooth	Plugged Pito Tube	4	4	5	80		4	4	5	80
				High Angle of Attack	4	4	2	32		4	4	2	32
				Incorrect Mounting	4	2	2	16		4	2	2	16
		Hardware Failure	Flight Less Smooth	Poorly Connected Electrical Joint	4	1	7	28		4	1	7	28
Inertial Sense	Measure acceleration, barometter data, and magnetic heading	Software Failure	Crash	Internal Code	9	1	10	90	Extensive testing prior to use	9	1	3	27
		Inaccurate Readings	Crash	Interference	9	3	8	216	Extensive testing prior to use	9	3	3	81
		Hardware Failure	Crash	Poorly Connected Electrical Joint	9	1	7	63	Extensive testing prior to use	9	1	3	27
GPS	Measure global position	Software Failure	Crash	Internal Code	9	3	10	270	Extensive testing prior to use	9	3	3	81
		Inaccurate Readings	Crash	Interference	9	4	5	180	FFCL	9	4	4	144
		Hardware Failure	Mission Failure Manual Landing	Poorly Connected Electrical Joint	6	1	7	42		6	1	7	42
Battery	Provide current to all systems in the air	Loss of Power	Crash	Battery Not Charged Correctly	9	5	3	135	FFCL	9	5	2	90
				Chemical Misshap	10	2	3	60	Assign battery saftey officer	10	1	2	20
				Battery Degridation	9	1	1	9	FFCL	9	1	1	9
ESCs	BEC and convert digital logic PWM to high voltage/current motor inputs	Hardware Failure	Crash	Poorly Connected Electrical Joint	9	1	7	63	Extensive testing prior to use	9	1	3	27
Motors	Rotate Props	Overheat	Fire and Crash	Overstressing the Motors	10	3	5	150	Add warning to FFCL	10	2	5	100
		Does Not Transmit Tourque	Mission Failure Glide to Safe Landing	Props Unsecured	7	8	3	168	FFCL	7	5	2	70
		Rotates the Wrong Way	Mission Does Not Start	Wires Connected Backwards	6	3	2	36	FFCL	6	1	2	12
		Hardware Failure	Mission Failure Glide to Safe Landing	Poorly Connected Electrical Joint	7	1	7	49		7	1	7	49
Props	Provide Thrust	Does Not Provide Thrust	Mission Failure Glide to Safe Landing	Chipped/broken prop	7	5	3	105		7	5	3	105
Wiring	Transmit power and signals	Provides Electricty to Incorrect Location	Crash	Wires Connected to Incorrect Ports	9	7	8	504	FFCL	9	3	3	81
		Does Not Transmit Electricity	Crash	Electrical Short Circuit	9	3	8	216	Shrink wrap all exposed wires	9	1	8	72
			Crash	Electrical Open Circuit	9	8	5	360	FFCL	9	8	1	72
Servos	Move control surfaces	Linkage Breaks	Crash	Poorly Assembled	9	2	7	126	Extensive testing prior to use	9	2	5	90
				Large Controll Inputs at High Velocity	9	1	3	27	Train saftey pilot	9	1	3	27
		Mechanical Limits Exceeded	Crash	Aerobatic Flight Saturates Controller	9	5	8	360	Train saftey pilot	9	1	4	36
				Poorly Assembled	9	6	4	216	Extensive testing prior to use	9	2	4	72
		Software Failure	Crash	Internal Code	9	1	10	90	Extensive testing prior to use	9	1	3	27
		Hardware Failure	Crash	Poorly Connected Electrical Joint	9	1	7	63	Extensive testing prior to use	9	1	3	27
		Internal Mechanics Broken	Crash	Overuse	9	2	5	90		9	2	5	90
		Servo Burns Out	Crash	Overuse	9	2	5	90		9	2	5	90
UGV System	Deliver water bottle to both ground locations	See UGV Documentation for UGV FMEA											
Imaging System	Capture, interperate, and report ground targets	See Imaging Documentation for Imaging FMEA											
Control Software	Pilot aircraft autonomusly	See Control Documentation for Control FMEA											
Communication Software	Allow communication of all components	See Communication Documentation for Communication FMEA											
Airframe Body	Contain components, provide lift, provide stability, & respond to control inputs	Flight Characteristics Change	Crash	Icing	9	2	1	18	Only fly in good weather	9	1	1	9
				Components Move	9	5	5	225	Strap down all components	9	3	3	81
		Parts Break Off	Crash	Flight Envelop Exceeded	9	2	3	54	Train saftey pilot	9	2	2	36
				Poor Manufacturing	9	6	7	378	Extensive testing prior to use	9	6	2	108
				Part poorly attached	9	2	7	126	FFCL	9	2	3	54
				Unidentified Flying Object (UFO) Impact	9	1	3	27	Train saftey pilot	9	1	3	27
Ground stations	Transmit high level commands between operators and WIFI router	Battery Dies	Mission Failure Manual Landing	Charger Not Connected	6	1	1	6		6	1	1	6
		Hardware Failure	Mission Failure Manual Landing	Poorly Connected Electrical Joint	6	1	7	42		6	1	7	42
		Software Failure	Crash	Bug in Code	9	7	10	630	Extensive testing prior to use	9	4	3	108
WIFI Router	Trasmit data over ROS network between groundstations to light beam	Loss of Connection	Mission Failure Manual Landing	Interference	6	2	7	84		6	2	7	84
		Hardware Failure	Mission Failure Manual Landing	Poorly Connected Electrical Joint	6	1	7	42		6	1	7	42
		Software Failure	Mission Failure Manual Landing	Internal Code	6	1	10	60		6	1	10	60
WIFI Light Beam	Transmit data over ROS network between WIFI router and the WIFI antenna on the aircraft	Loss of Connection	Mission Failure Manual Landing	Interference	6	8	7	336	Perform BPS range test	6	5	7	210
		Hardware Failure	Mission Failure Manual Landing	Poorly Connected Electrical Joint	6	1	7	42		6	1	7	42
		Software Failure	Mission Failure Manual Landing	Internal Code	6	1	10	60		6	1	10	60
Ground Power Source	Provide current to all ground systems	Not Brought with Us	Mission Does Not Start	Poor Planning	4	8	4	128	FFCL	4	4	4	64
		Mechanical Failure	Mission Failure Manual Landing	Poor Manufacturing	6	1	7	42		6	1	7	42
Human Operators	Give high level commands & ensure saftey of flight	Sick	Mission Does Not Start	Bacteria or Viruses	5	4	3	60		5	4	3	60
		Can Not Attend	Mission Does Not Start	Other Plans	5	1	1	5		5	1	1	5
		Sends Incorrect Commands	Crash	Poor Judgement	9	2	9	162	Extensive practice	9	1	9	81
			Crash	Poor Understanding of System	9	2	5	90	Extensive practice	9	1	5	45

* In this analysis "Hardware Failure" refers only to electrical hardware (e.g. USB port breaks or soldering fails)

** FFCL is the Field Flight Checklist to which we will add items to test and do before flight

*** Extensive testing before use refers to extensive flight tests before the competition.
We currently perform flight tests a couple times a week.

S: Severity of failure effect

L: Likelihood of failure occurring

D: Dectability of cause before failure occurs

RPN: Risk Priority number (S*L*D)

3 Discussion

As can be seen from this analysis, most of the concerning issues were addressed. We are now confident in our ability to fly a failure free mission with the exception of one issue: we continue to see communication drop out for a couple systems. We do not completely understand why this is happening. It seems to be location dependent and caused by interfering signals. It does not usually affect our missions, but it's risk priority number (RPN) is high enough that we would like to address it. We will perform tests in several locations to see if we can identify the root cause and solution to these communication issues. We would like to be confident that this issue will not arise at the competition.

Brigham Young University

AUVSI 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. We will devote time and other resources to other aspects of the project.

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 CG placement 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**
y(m)	[0.000 0.925 0.975]
chord(m)	[0.310 0.250 0.250]
offset(m)	[0.000 0.000 0.000]
dihedral(°)	[0.000 0.000 0.000]
twist(°)	[0.000 0.000 5.000]
foil	NACA 3311

Table 2: XFLR5 Elevator Parameters

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

Table 3: XFLR5 Fin Parameters

Parameter (units)	**Value**
y(m)	[0.000 0.270]
chord(m)	[0.220 0.120]
offset(m)	[0.000 0.110]
dihedral(°)	[0.000 0.000]
twist(°)	[0.000 0.000]
foil	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.

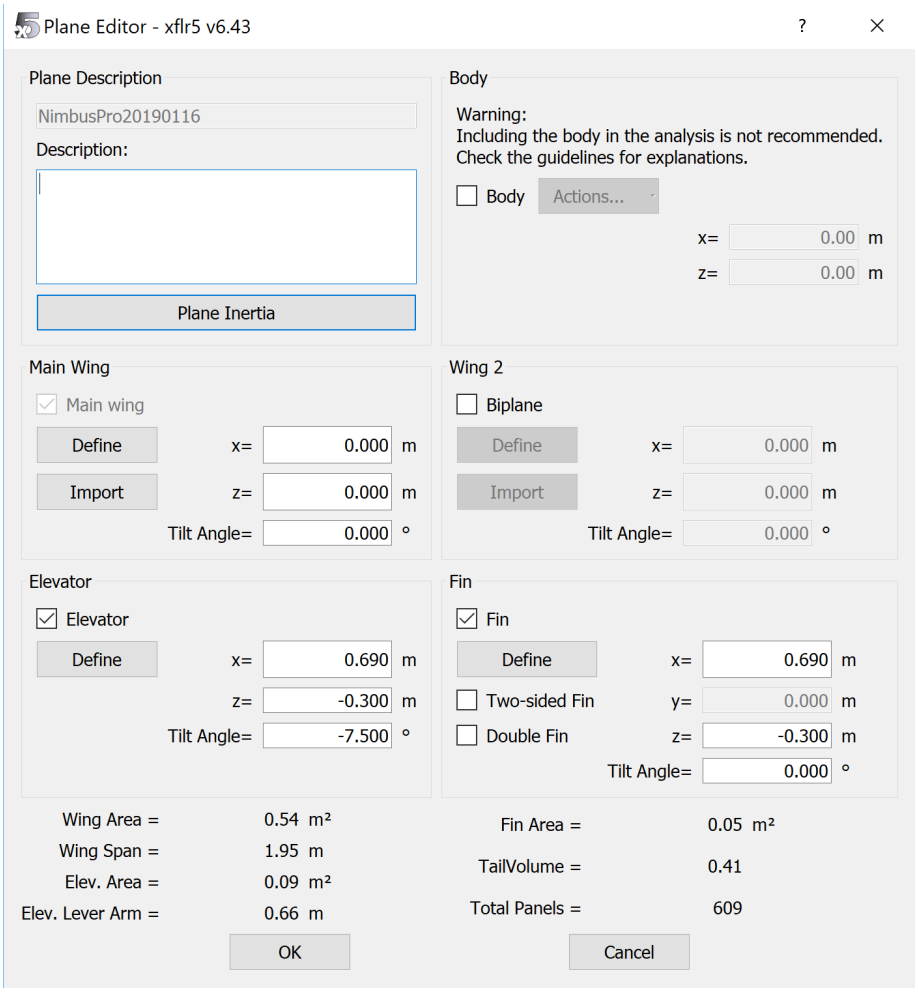


Figure 1: XFLR5 dialog summarizing wing, fin, and elevator parameters.

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.
- α ranging from $-8.0°$ to $18.0°$ at $0.25°$ 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.

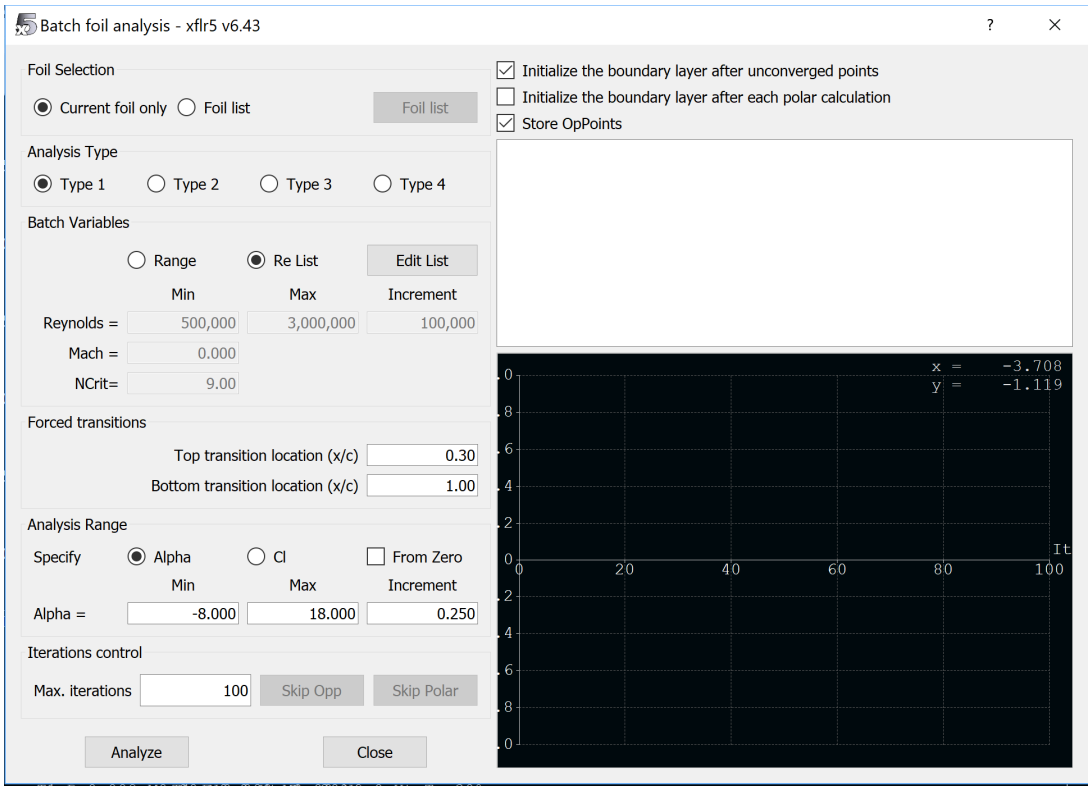


Figure 4: Dialog window for performing an airfoil batch analysis.

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

- $\rho = 1.225kg/m3$
- $\nu = 1.5e-05m^2/s$
- Ground Effect: NO

Extra Drag

- Extra area(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 = $-5.000°$

- End = 15.000°

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
- $\beta = 0.00°$
- $\phi = 0.00°$

Ref. dimensions

- Wing Planform projected on xy plane

Mass and inertia

- Use plane inertia: YES

Control parameters

- ALL ZERO

Aero data

- $\rho = 1.225kg/m3$
- $\nu = 1.5e - 05m^2/s$
- Ground Effect: NO

Extra Drag

- Extra area(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 $9.2°$ and reaches a maximum at approximately $13°$. Since we have about $5°$ of wash-in in the wing design, it is desirable to fly at an angle of attack of $\alpha < 13° - 5° = 8°$ 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.

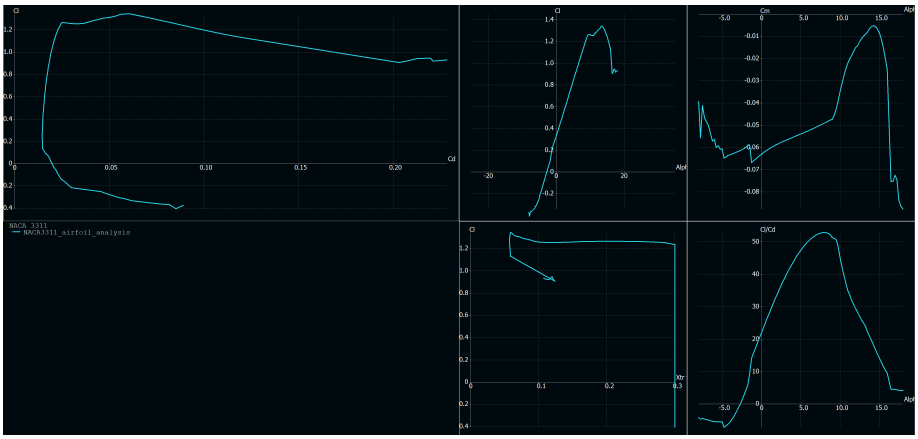


Figure 5: Drag polars of the main wing airfoil at design speed conditions.

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. α 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. α 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° was selected, about 1° 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 Cm vs. α plot) was obtained. Because the Cm vs. α 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° 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°.

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. α 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°. This resulted in a design angle of attack of about 7°, 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°. See AF-10 for the tail modification design artifact.

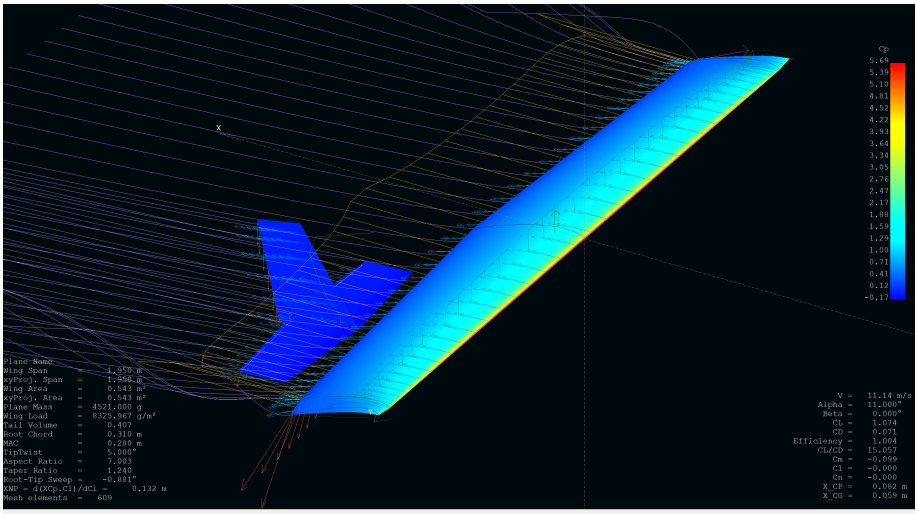
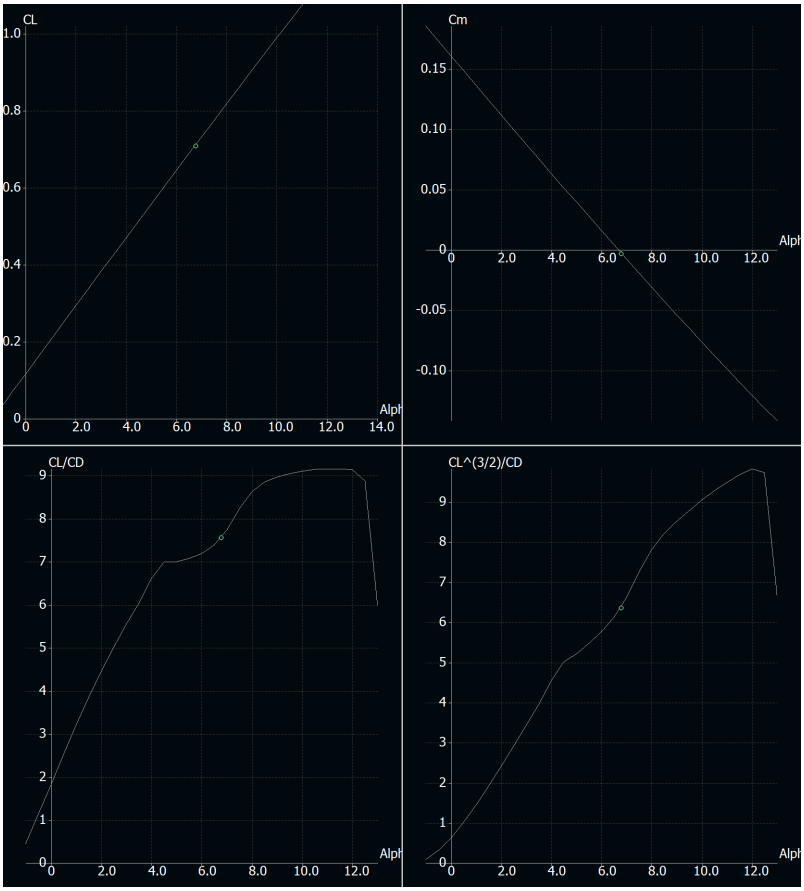


Figure 6: Rendering of the analyzed plane.



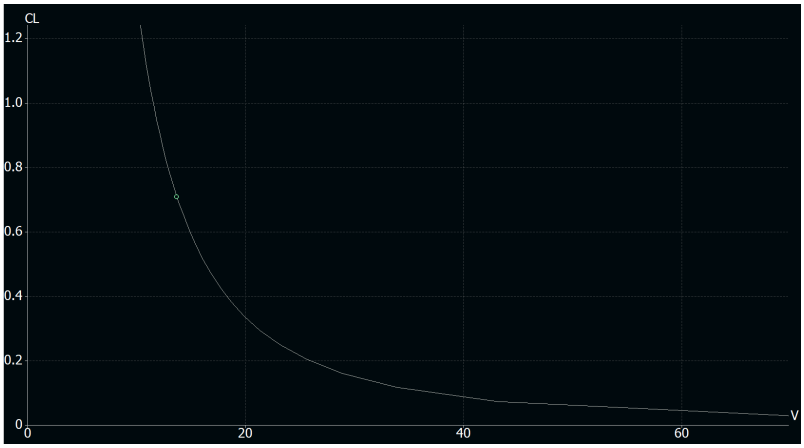


Figure 7: Drag polars of the plane.

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:

- $Cnb > 0$ indicating static directional stability (yaw axis).
- $Cma < 0$ indicating static longitudinal stability (pitch axis).
- $Clb > 0$ 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

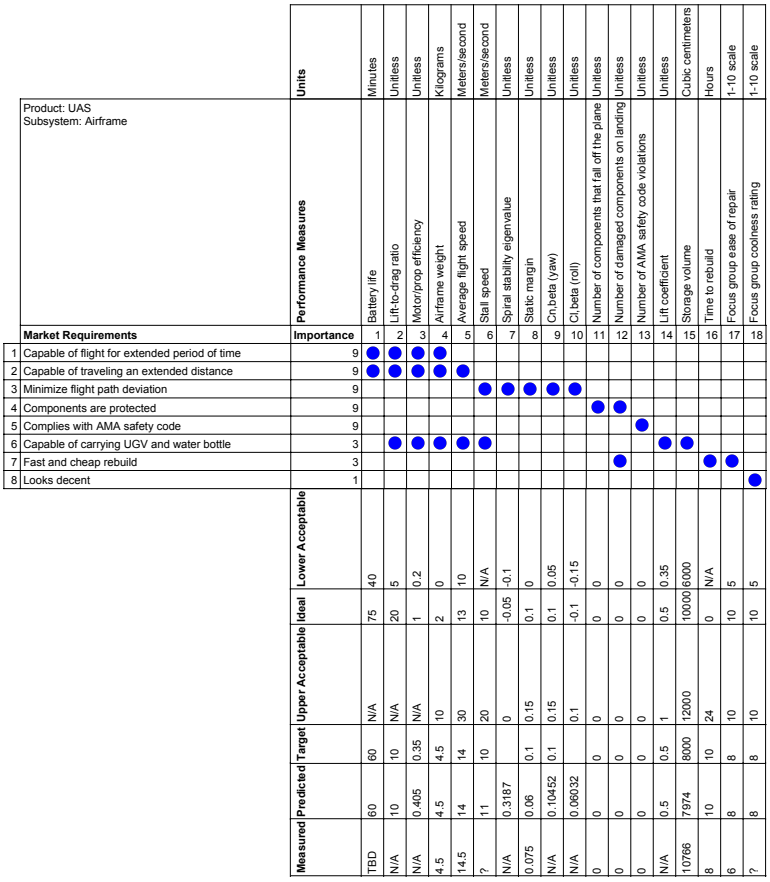
AUVSI Capstone Team (Team 45)

Airframe Subsystem Requirements Matrix

ID	Rev.	Date	Description	Author	Checked By
AF-001	0.1	10-23-18	Initial Draft	Tyler Critchfield & Ryan Anderson	Derek Knowles
AF-001	0.2	11-06-18	Concept Development	Tyler Critchfield	Ryan Anderson & Kameron Eves
AF-001	1.1	2-12-19	Subsystem Engineering	Tyler Critchfield	CHECKED BY

1 Introduction

Figure 1 shows our updated Requirements Matrix for the Airframe subsystem. Section E has been added with the Target, Predicted, and measured values for our performance measures. Some measures were determined in models (see Artifact ——) and could not be empirically measured. As such, these values were placed in the predicted row, and N/A was assigned to the measured row.



	Product: UAS Subsystem: Airframe	Units	Minutes	Unitless	Unitless	Kilograms	Meters/second	Meters/second	Unitless	Unitless	Unitless	Unitless	Unitless	Unitless	Unitless	Unitless	Unitless	Cubic centimeters	Hours	1-10 scale	1-10 scale
	Market Requirements	**Performance Measures**	Battery life	Lift-to-drag ratio	Motor/prop efficiency	Airframe weight	Average flight speed	Stall speed	Spiral stability eigenvalue	Static margin	Cn,beta (yaw)	Cl,beta (roll)	Number of components that fall off the plane	Number of damaged components on landing	Number of AMA safety code violations	Lift coefficient	Storage volume	Time to rebuild	Focus group ease of repair	Focus group coolness rating
		Importance	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
1	Capable of flight for extended period of time	9	●	●	●	●														
2	Capable of traveling an extended distance	9	●	●	●	●	●													
3	Minimize flight path deviation	9						●	●	●	●	●								
4	Components are protected	9											●	●						
5	Complies with AMA safety code	9													●					
6	Capable of carrying UGV and water bottle	3		●	●	●	●	●								●	●			
7	Fast and cheap rebuild	3												●				●	●	
8	Looks decent	1																		●
		Lower Acceptable	40	5	0.2	0	10	N/A	-0.1	0	0.05	-0.15	0	0	0	0.35	6000	N/A	5	5
		Ideal	75	20	1	2	13	10	-0.05	0.1	0.1	-0.1	0	0	0	0.5	10000	0	10	10
		Upper Acceptable	N/A	N/A	N/A	10	30	20	0	0.15	0.15	0.1	0	0	0	1	12000	24	10	10
		Target	60	10	0.35	4.5	14	10		0.1	0.1		0	0	0	0.5	8000	10	8	8
		Predicted	60	10	0.405	4.5	14	11	0.3187	0.06	0.10452	0.06032	0	0	0	0.5	7974	10	8	8
		Measured	TBD	N/A	N/A	4.5	14.5	?	N/A	0.075	N/A	N/A	0	0	0	N/A	10766	8	6	?

Figure 1: The updated requirements matrix for the airframe subsystem, with section E included (target, predicted and measured values for performance measures.)

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

Introduction

During our XFLR analysis detailed in AF-11 we determined that 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 provided negative lift to push the tail down 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) the 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 tradeoffs 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 aircrafts over all control scheme. The rudder is usually used to mitigate sideslip and allow for coordinated turns, but the 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 it's 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° 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 manufacture 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, cheep, light weight, and would not have to bare 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 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 were we trimmed the aircraft.

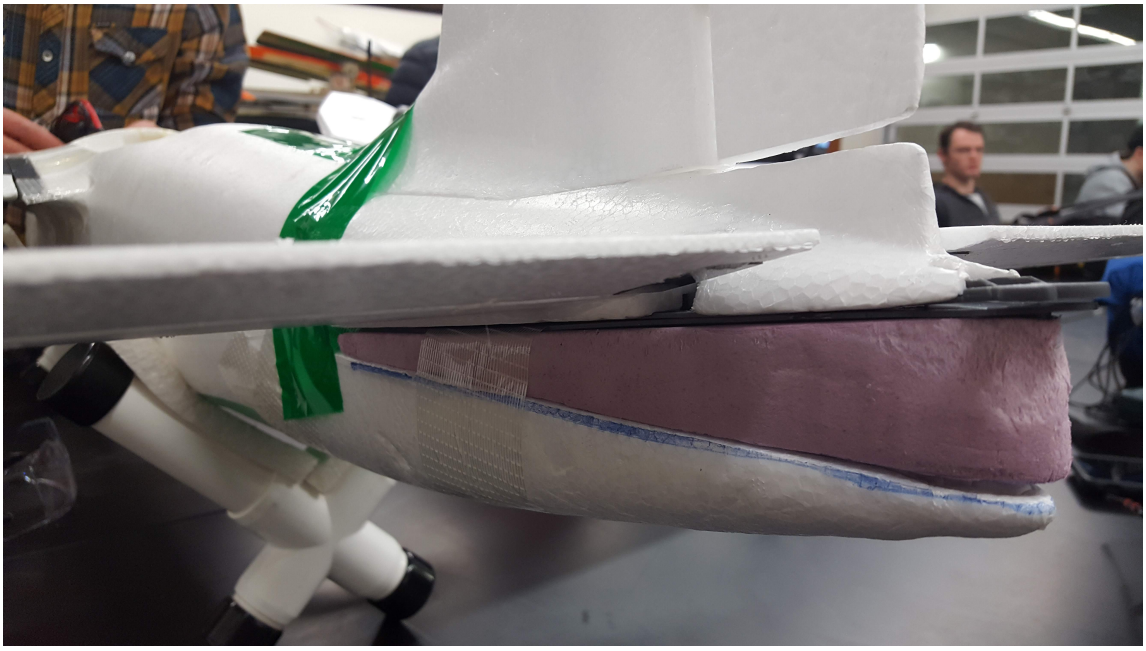


Figure 1: "The wedge finished and installed on the aircraft. Note that the outer surface of the wedge is flush with the outer surface of the aircraft's tail."

Conclusion

We are very pleased with the implementation of this wedge as it's success it 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 measure to be achieved. More specifically, the increased control authority will also 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. Both of which are key success measures.

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

Brigham Young University

AUVSI Capstone Team (Team 45)

Airframe Testing Procedures

ID	Rev.	Date	Description	Author	Checked By
AF-013	0.1	2/21/19	Created	Kameron Eves	Tyler Critchfield

1 Introduction

Several tests were performed to ensure that our design, when actually implemented, meets the requirements and key success measures. A description of these tests can be found below. The results of these tests are included in the requirements matrix (See AF-001).

2 Tests

- **Requirement:** Test Description
- **Battery Life:** Battery voltage was measured before and after flight. Voltage was assumed to vary linearly from the charged voltage and discharged voltage.
- **Lift-to-Drag Ratio:** While in flight we set the throttle to zero and let the aircraft glide. We noted the start and end time of the glide. Because the aircraft's glide ratio is mathematically equal to the Lift-to-Drag Ratio we then could use a global positioning system and altitude measurements from a barometer to determine the Lift-to-Drag Ratio. Appendix 1 is the code for this calculation.
- **Motor/Prop Efficiency:** Motor and prop parameters were input into code written by Dr. Andrew Ning, a professor of mechanical engineering at Brigham Young University's Mechanical Engineering department. This code takes the motor and prop parameters and returns, among other things, the efficiency of the motors. Contact Dr. Ning for a copy of this code.
- **Airframe Weight:** The aircraft was loaded with all of the components that will be used in the competition, including the payload. The aircraft was then placed on a digital scale and the weight read from the scale's reading.
- **Average Flight Speed:** Average flight speed was found by averaging the velocity of the aircraft over a flight as read by a differential pressure sensor and a pitot tube.
- **Stall Speed:** Stall speed was found by manually piloting the aircraft as close to stall speed as we could without stalling and then reading the velocity of the aircraft from a differential pressure sensor and a pitot tube.
- **Spiral Stability Eigenvalue:** Numerical analysis. See AF-011.
- **Static Margin (with Payload):** The location of the CG was measured by placing the aircraft on a balancing apparatus. The aerodynamic center was assumed to be at the quarter chord of the wing. The static margin is then the difference between these two values normalized by the wing's mean chord length.

- **$C_{n,beta}$ (yaw):** Numerical analysis. See AF-011.
- **$C_{l,beta}$ (roll):** Numerical analysis. See AF-011.
- **Number of Components that Fall of the Plane:** The number of components that fell off of the aircraft over the previous 10 flights was averaged to obtain the average number of components that fall of the aircraft per flight.
- **Number of Damaged Components on Landing:** The number of components that damaged over the previous 10 flights was averaged to obtain the average number of components that damaged per flight.
- **Number of AMA Safety Code Violations:** The AMA safety codes were read with the our aircraft and design in mind. The number of potential violations was then summed.
- **Lift Coeffcient:** Numerical analysis. See AF-011.
- **Storage Volume:** The aircraft was loaded with all components to be used in the competition except the payload and payload drop mechanism. The dimensions available for these payload components were then measured with a ruler and the volume computed by multiplying the dimensions together.
- **Time to Rebuild:** We had one catastrophic crash with the Nimbus Pro. This is the approximate time it took to get the plane ready to fly again.
- **Focus Group Ease of Repair:** Several team members were enlisted to help repair the aircraft after several flight tests. Their responses to the question, "On a scale of 1 to 10 with 10 being the easiest, how easy would you say is was to repair this aircraft?" Respondents were encouraged to use previous RC experience as a frame of reference.

3 Conclusion

As can be seen in the above discussion we thoroughly tested our aircraft to ensure it met the market requirements. Because of these tests we are confident that our aircraft meets and even exceeds these requirements.

Brigham Young University

AUVSI Capstone Team (Team 45)

Autopilot Testing Procedures and Results

ID	Rev.	Date	Description	Author	Checked By
CT-002	0.1	02-12-2019	Initial conception	Andrew Torgesen	Jacob Willis

1 Introduction

The autopilot for the UAS is called **ROSPlane**. It is responsible for taking high-level flight path commands and translating them to low-level actuator commands (aileron and elevator servos and throttle) on the airframe. The autopilot we are using and its software architecture are documented in our team Github repository.

An intermediate step for the UAS to achieve its key success measures is to ensure that the underlying autopilot is well-tuned. The phrase well-tuned refers to the fact that the autopilot consists of a series of PID loops to control the longitudinal and lateral autopilots. Each PID loop has associated gains which must be tuned in-flight to ensure optimal performance. The following are block diagrams of ROSPlane's inner and outer loops for the longitudinal and lateral autopilots, respectively:

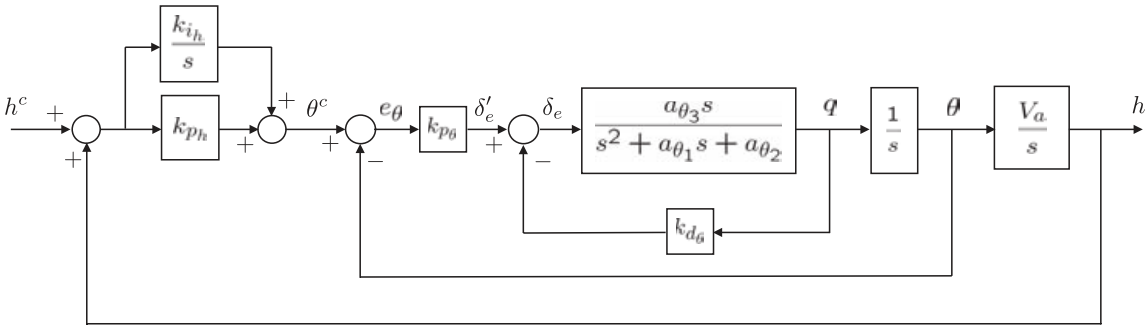


Figure 1: Longitudinal successive loop closure autopilot for the UAS. Borrowed from Small Unmanned Aircraft - Theory and Practice by Beard, McLain.

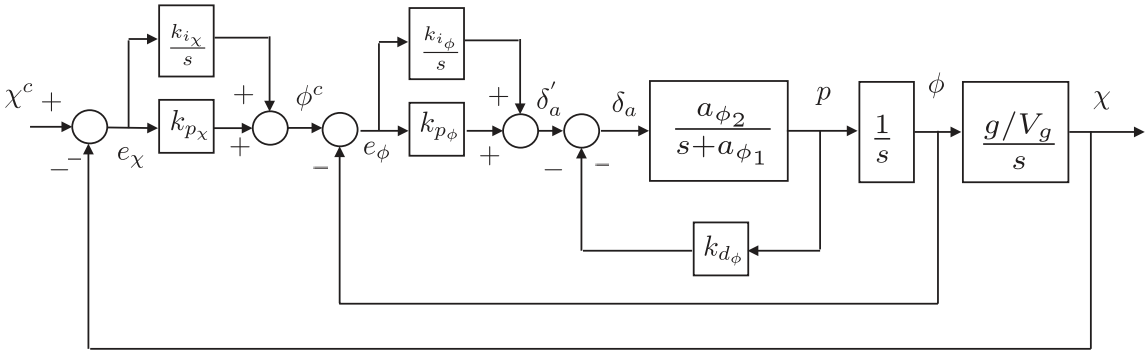


Figure 2: Lateral successive loop closure autopilot for the UAS. Borrowed from Small Unmanned Aircraft - Theory and Practice by Beard, McLain.

This artifact details our testing and evaluation procedures to ensure that autopilot performance and compatibility with the rest of the UAS.

2 Performance Testing

Performance testing of ROSPlane is divided into two parts: control algorithm testing and estimation algorithm testing. The control algorithms constitute the functionality of the autopilot, and thus must be well-tuned to ensure that the key success measures can be met. Moreover, the estimation algorithms (currently being run on the Inertial Sense hardware–see the *UAS Subsystem Interface Definition* artifact (SS-001)) provide vital information about the current dynamic state of the UAS to the autopilot, and thus must also be validated to ensure stable unmanned flight.

2.1 Control Algorithms

Control algorithm testing consists of determining how well the autopilot is able to follow commanded flight path states, such as pitch angle, altitude, roll angle, and course angle. Good performance entails timely convergence to the commanded values without subsequent oscillation or instability. To ensure good performance according to these criteria, we followed the procedures outlined in the *Autopilot Tuning* artifact (CT-001) over the course of three different flight tests. Below are plots from real flight test data demonstrating the ability of the autopilot to converge on the commanded flight path states of altitude (h) and course angle (χ):

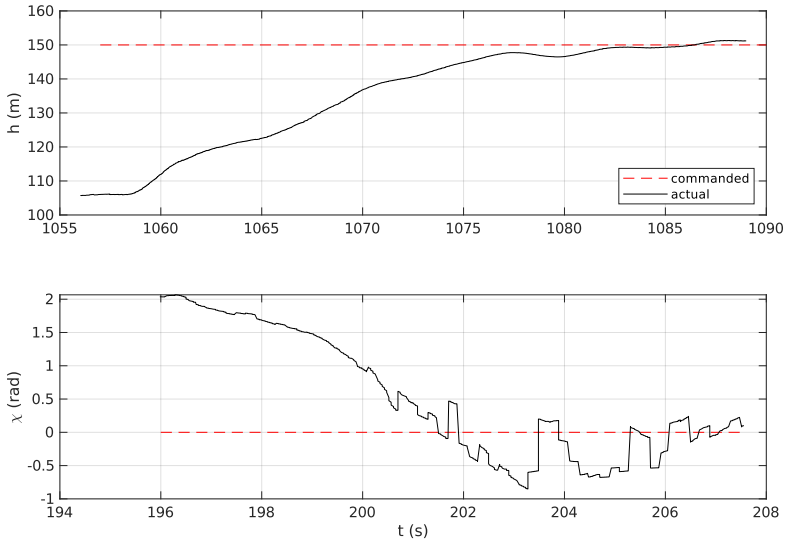


Figure 3: Flight test data demonstrating the ability of the autopilot to converge on commanded values for altitude and course angle.

As can be seen from Figure 3, the longitudinal controller converges smoothly, whereas the lateral controller appears to struggle a bit to converge. This is due to the fact that there was a bit of wind blowing from West to East the day of our flight testing. Nevertheless, the lateral controller has demonstrated the ability to overcome external wind disturbances and arrive at the commanded course angle value, which was due North in this case.

One final thing to note from Figure 3 is the chopiness of the measured course angle data. This choppiness is attributed to state estimator error from our onboard sensors, which is addressed in the following section.

2.2 Estimation Algorithms

Through flight testing and subsequent analysis of the estimated state data, we have determined that using the Inertial Sense sensor module for state estimation alone, while basically adequate for unmanned flight, is too subject to small failures which propagate into large problems for the UAS as a whole. The following are the two principal issues with the Inertial Sense Estimation that we have observed:

2.2.1 Heading Estimation

As can be seen from Figure 3, the course angle data from the Inertial Sense can be choppy. The Inertial Sense is known to have issues estimating yaw if it isn't moving, but still apparently sometimes suffers from large amount of noise while moving. This failure to produce a smooth course angle measurement could conceivably cause issues for the control algorithms, and thus must be amended.

2.2.2 Altitude Estimation

Figure 4 demonstrates another weakness with the Inertial Sense that occasionally arises. After $t \approx 75s$, the altitude estimate begins to go negative, and fails to recover. This behavior is attributable to the fact that the sensor is relying on GPS and inertial data only to estimate altitude. This failure to estimate altitude accurately is catastrophic when it arises, as the autopilot has no accurate idea of where it really is, leading to undesirable and unpredictable behavior.

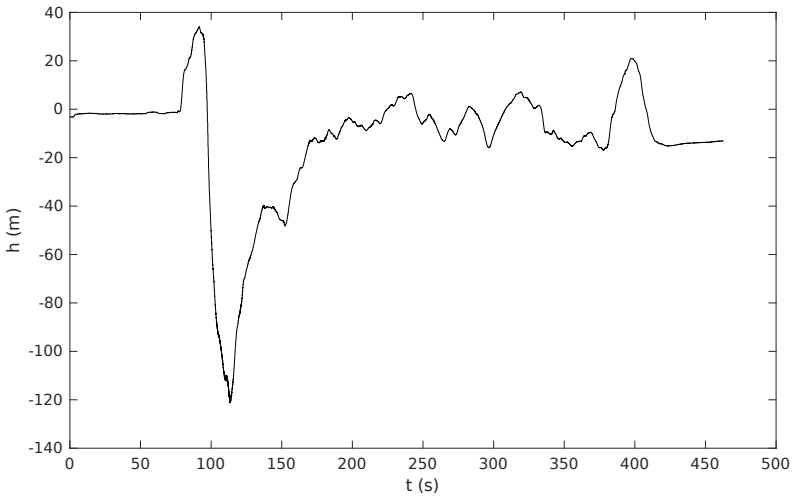


Figure 4: Altitude estimation output from the Inertial Sense sensor during a test flight.

2.2.3 Plans for Addressing Issues

To avoid the failure modes in course angle and altitude estimation discussed above, our plan is to leverage the sensor fusion capabilities of the ROSFlight board and ROSPlane to add sensor redundancy. ROSFlight and ROSPlane together constitute an estimation scheme that is specially designed for fixed-wing platforms, and they integrate sensors that the Inertial Sense does not, such as an airspeed sensor and barometer. It is anticipated that this added redundancy will greatly reduce the risk of state estimation failure due to experiences in previous years. That being said, extensive testing of the redundant system will still be carried out.

3 System Compatibility Testing

The *UAS Subsystem Interface Definition* artifact (SS-001) details that there are four interfaces between ROSPlane and other components in the UAS:

1. Flight Controller/Inertial Sense Interface (*two-way*)
2. Image Stamper Interface (*two-way*)
3. Ubiquiti Bullet Interface (*two-way*)
4. Payload Delivery Interface (*one-way*)

Table 1 communicates our testing procedures and results for each of these interfaces.

Table 1: Description of testing procedures and results for ROSPlane interfaces.

Interface	Testing Procedure	Testing Results
Flight Controller/Inertial Sense Interface	This interface runs over a USB cable using the MAVLink protocol. The MAVLink protocol itself is a tried-and-true protocol for serial communication between devices. Our particular MAVLink connection has been repeatedly tested on each flight test, whose procedure is outlined in the *Field Flight Checklist* artifact (PF-001). A working flight controller interface is required for both RC and unmanned flight.	The *Flight Test Log* artifact (AF-004) details that over the course of 13 flight tests, the flight controller interface has never posed a problem. That being said, the *Estimation Algorithms* section of this artifact details the weaknesses of the data being passed to the autopilot from the Inertial Sense hardware.
Image Stamper Interface	This interface runs over ROS, following the publisher-subscriber architecture. Testing this architecture has entailed running the ROSPlane and image stamper nodes, having each node publish messages to each other, and ensuring a constant message reception frequency in each node.	ROS network testing has shown that publisher-subscriber connectivity has never posed an issue, particularly because both nodes are running on the Odroid computer and thus not dependent on a reliable WiFi connection.
Ubiquiti Bullet Interface	Testing of this interface entails ensuring relatively constant connectivity over WiFi from the ground to the plane during flight testing. It is useful to use the *ping* command to check connection speed throughout the flight.	Over the course of several flight tests, only once has WiFi connectivity posed an issue. This was most likely attributable to failing to keep the Ubiquiti Rocket pointed at the plane constantly during flight.

Payload Delivery Interface	The payload delivery mechanism is last year's design, and their Subsystem Engineering Artifacts may be consulted for details on this interface.	A summary of the results is that the interface is reliable; the only problems with the payload delivery system are due to the behavior of the Odroid's operating system itself when ROS is not running.

4 Conclusion

Extensive testing of the autopilot subsystem, primarily in the form of flight tests and autopilot tuning, has determined that the autopilot performs up to standards and is capable of interfacing with the needed components in the UAS. Further work is needed to improve the robustness of the onboard state estimation to greatly reduce the risk of in-flight failure, since ROSPlane relies on accurate state estimation to sustain unmanned flight.

Brigham Young University

AUVSI Capstone Team (Team 45)

Autopilot Tuning

ID	Rev.	Date	Description	Author	Checked By
CT-001	0.1	01-23-2019	Initial conception	Andrew Torgesen	Brandon McBride

1 Introduction

Our chosen fixed-wing autopilot for the UAV is ROSPlane, an open source implementation of the control, estimation, and path planning algorithms set forth in *Small Unmanned Aircraft: Theory and Practice* by Randal Beard and Timothy McLain. This artifact sets forth the general process for tuning the proportional, integral, and derivative (PID) gains for the autopilot, based on both theory from the text as well as personal experience.

2 Description of the Gains

Trim Gains:

- Elevator trim
- Aileron trim
- Rudder trim
- Throttle trim

These gains must be written to ROSFlight before turning on the autopilot for the first time. The directions for doing this are found at http://docs.rosflight.org/en/latest/user-guide/performance/ under the heading **RC trim (Feed-Forward Torque Calculation)**.

Longitudinal Autopilot Gains:

- Pitch_Kp
- Pitch_Kd
- Pitch_Ki
- ALT_KP
- ALT_KD
- ALT_KI
- AS_PITCH_KP
- AS_PITCH_KD
- AS_PITCH_KI
- AS_THR_KP

- AS_THR_KD
- AS_THR_KI

The suffixes Kp, Kd, and Ki refer to proportional, derivative, and integral gains, respectively. The Pitch gains control the inner loop of the longitudinal autopilot, and their tuning should correspond to a desirable convergence of current pitch to commanded pitch. The ALT gains are outer loop gains that determine the ability to hold an altitude without excessive deviation in the presence of disturbances.

In general, the AS_PITCH_ gains determine the aircraft's ability to hold an airspeed while landing, during which time pitch commands are used to control speed. Conversely, the AS_THR_ gains determine the aircraft's ability to hold an airspeed while flying, during which time throttle commands are used to control speed. The aircraft should be able to hold an airspeed even in the presence of wind and other external disturbances.

Lateral Autopilot Gains:

- Roll_Kp
- Roll_Kd
- Roll_Ki
- Course_Kp
- Course_Kd
- Course_Ki
- BETA_KP
- BETA_KD
- BETA_KI

The Roll gains control the inner loop of the lateral autopilot, and their tuning should correspond to a desirable convergence of current roll angle to commanded roll. The Course gains are outer loop gains that determine the ability to hold a commanded course angle without excessive deviation in the presence of disturbances. The BETA gains refer to slide-slip holding–in general, we don't worry too much about those, especially since we are not using rudder control.

Path Follower Gains:

- CHI_INFTY
- K_PATH

- K_ORBIT

These are gains that are tuned in accordance with the aircraft's ability to make tight turns of different radii and speeds. K_PATH determines how fast the plane approaches the straight line desired path. A high value causes the plane to fly perpendicular to the line until it reaches it while a low value causes the plane to asymptotically approach the desired line. CHI_INFTY is the course angle at which the path follower will approach a path if at an infinite distance away. K_ORBIT determines how fast a plane transitions from a straight line to an orbit. While the plane is far away from the center of the loiter locaiton, it will fly directly to the center but when the plane is close to the loiter location it flies the orbit path. A low K_ORBIT value causes the plane to have a more gradual transition from straight line to orbit.

3 Sequence of Gain Tuning (Practical Guide)

Note: The language used in this guide assumes at least a working knowledge of the Robot Operating System (ROS). ROS tutorials may be found at http://wiki.ros.org/ROS/Tutorials.

While running ROSPlane on a ROS network, the *rqt_reconfigure* plugin allows the user to view (and modify in real-time) all configurable gains for the autopilot. Figure 1 is a screenshot of the configuration user interface:

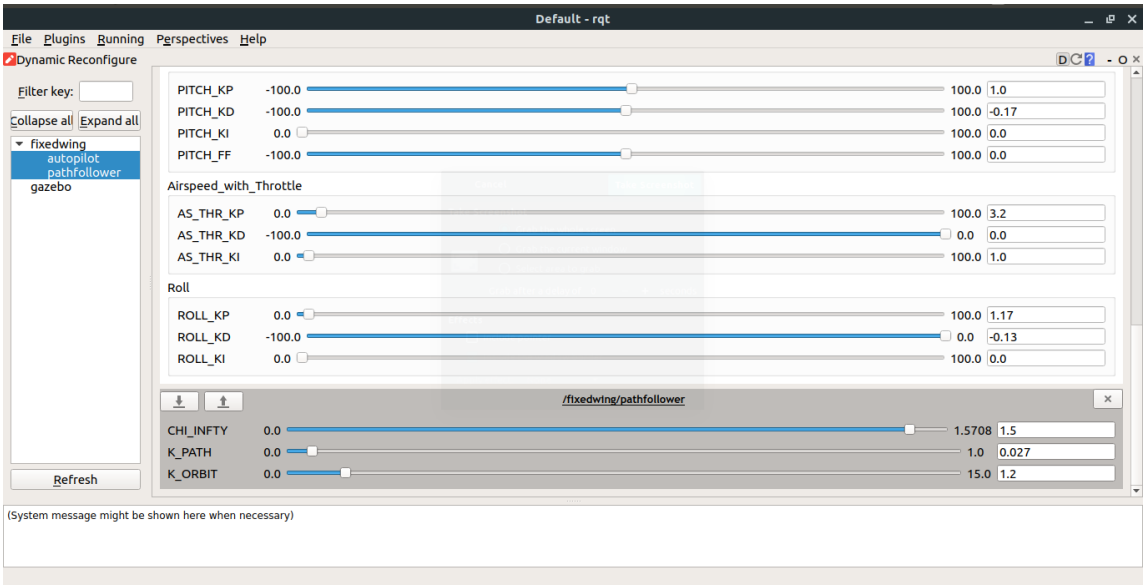


Figure 1: Dynamic gain reconfiguration user interface.

Before proceeding, keep the following heuristic for tuning PID gains in mind:

1. Increase P until the response time is satisfactory, albeit with a little bit of overshoot/oscillation.
2. Increase D until overshoot/oscillation is removed.
3. Increase I *only if* there is steady-state error.
4. **Iterate.**

3.1 Tune the Longitudinal Autopilot

Kill the path follower node with the command *rosnode kill /pathfollower*. Use the command *rqt_graph* to ensure that the node is really terminated once you've attempted to kill it. With the path manager killed and the reconfigure window open, send a command to the path follower to hold a pitch and altitude. For example, we sent the following command to the autopilot, which had the autopilot flying at an airspeed of 15 m/s, an altitude of 70 m, and a northern heading:

rostopic pub -r 50 /controller_commands rosplane_msgs/Controller_Commands – 15 70 0.0 0.0 '[0.0, 0.0, 0.0, 0.0]' False False

With this command running, tune the longitudinal gains in the following order:

1. Pitch gains
2. Altitude hold gains
3. Airspeed hold gains

Successful gain tuning should have the autopilot successfully flying at your commanded altitude at a relatively constant airspeed. Try outputing a new command with a different airspeed and altitude and see how your autopilot responds; it should transition smoothly to the new commanded values.

3.2 Tune the Lateral Autopilot

Relaunch the autopilot nodes in a subsequent flight to bring the path follower node back up. Then kill the path manager node with the command *rosnode kill /pathmanager*. Use the command *rqt_graph* to ensure that the node is really terminated once you've attempted to kill it. Now, send a command to the path follower to hold a roll and course. For example, we sent the following command to the autopilot, which had the autopilot

flying at an airspeed of 12 m/s in a clockwise orbit of radius 75 m, origin (-100 m North, -100 m East), and an altitude of 50 m:

rostopic pub -r 50 /current_path rosplane_msgs/Current_Path – 0 12 '[-1, -1, -1]' '[-1, -1, -1]' '[-100, -100, -50]' 75.0 1 False False

With this command running, tune the lateral gains in the following order:

1. Roll gains
2. Course hold gains
3. Path follower gains

Successful gain tuning should have the autopilot successfully flying at your commanded orbit at a relatively constant airspeed. Try outputing a new command with a different orbit and see how your autopilot responds; it should transition smoothly to the new commanded values.

3.3 Save the Gains

ROSPlane will not automatically save your new gain values from the reconfiguration interface window. You need to record the values that you set, put them in a .yaml file, and load those new parameters into your controller node upon every ROS launch! Likewise, the path follower gains must be loaded into the path follower node upon every ROS launch.

4 Conclusion

With satisfactorily tuned gains, the plane can now safely be used to fly waypoint paths. Try out some fun waypoint paths and see how well the autopilot handles them, even in the presence of wind and other disturbances!

Brigham Young University

AUVSI Capstone Team (Team 45)

Path Planner Testing Procedures and Results

ID	Rev.	Date	Description	Author	Checked By
CT-003	0.1	02-21-2019	Initial conception	John Akagi	Andrew Torgesen

Purpose

The purpose of this artifact is to explain the test procedures used to verify that the path planner is working correctly.

Procedure

In order to test the path planner, 5 simulations were run. Each simulation had 10 obstacles and 5 waypoints, each of which were chosen randomly. The locations of the obstacles and waypoints were written to a file and then the path planner planned a path, starting from the origin, that attempted to fly through each waypoint and avoid each obstacle. Once the path was planned, the simulated drone would fly the path and the position data was continually written to a file. Once the simulation finished, the distances between the waypoints and vehicle and obstacle and vehicle are computed. These distances are used to determine if the vehicle was close enough to the waypoint and if the vehicle hit the obstacle. These values were then compared with the key success measures. For reference, an excellent rating is defined in the key success measures as no more than 1 obstacle hit and flying within 20 feet (6 m) of each waypoint.

Results

The results of the 5 simulations can be seen in Table 1. In the five simulations, the vehicle only hit a single obstacle and was able to get within 5 meters on all but one waypoint. These results place the vehicle in the excellent category as defined by the key success measures.

Table 1: Table of results showing the number of obstacles hit and the minimum distance between the UAV and each waypoint.

Test No.	Waypoint 1	Waypoint 2	Waypoint 3	Waypoint 4	Waypoint 5	Obstacles Hit
1	.750 m	.128 m	.092 m	.430 m	.688 m	0
2	1.058 m	.127 m	.180 m	.376 m	.328m	0
3	.602 m	.625 m	.158 m	.204 m	.122 m	0
4	.427 m	.351 m	.042 m	.018 m	69.7m	1
5	.152 m	.085 m	.011 m	1.11 m	.90 m	0

Conclusion

Overall, the path planning and path following algorithms work well. The vehicle was able to achieve an excellent rating, as defined by the key success measures. However, some work is still needed to determine why the plane had difficulty hitting the last waypoint on the 4th trial. However, since the simulations can be run repeatedly multiple times, this should be easy to determine. Although it is likely that performance will be degraded when the algorithms are running on the actual plane since more variables, unknowns, and disturbances will be present, the algorithms have been proven to work to the level desired.

Brigham Young University

AUVSI Capstone Team (Team 45)

Unmanned Ground Vehicle Bay Door Description

ID	Rev.	Date	Description	Author	Checked By
GV-006	1.0	02-19-2019	Initial release	Derek Knowles	Brandon McBride

1 Introduction

The ability to successfully drop the unmanned ground vehicle is integral to the success of both the unmanned ground vehicle score and the successful completion of the rest of the mission. If the unmanned ground vehicle got tangled inside the plane and was prevented from successfully dropping from our plane, we would receive zero points for our unmanned ground vehicle and it would jeopardize the stability of our airframe and possibly result in crashing our plane.

2 Purpose

The purpose of our bay door design is to eliminate the potential for the unmanned ground vehicle to snag on anything while exiting the plane. The bay door design must also open reliably, close quickly, and prevent unintended openings.

3 Design Selected

We designed a bay door with a front hinge and torsional spring to close the door quickly. The bay door also includes plastic sheeting along the sides and a dedicated hole for the parachute to prevent any snagging upon exit.

4 Definition

The bay door for dropping the unmanned ground vehicle was cut out from the bottom of the plane's fuselage as close to the front of the plane as possible to help the center of gravity.

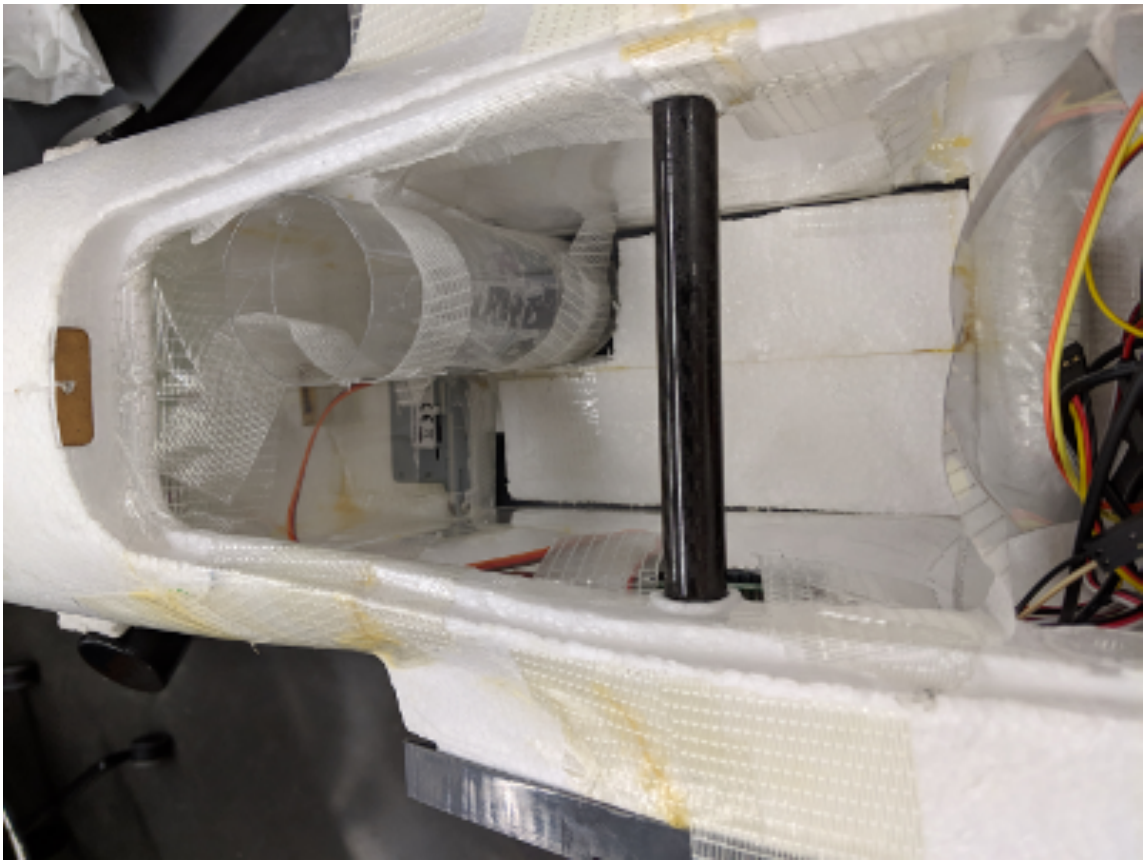


Figure 1: Unmanned ground vehicle bay door cut out

The release mechanism is a servo-less payload release that pulls a pin out of a latch attached to the bay door. The latch on the bay door is a thin acrylic piece attached to the middle of the bay door in order to increase its mechanical strength.

Figure 2: Torsional spring to assist closing the bay door

The front hinge includes a torsional spring so after deployment the bay door returns to the closed position.

Figure 3: Unmanned ground vehicle and parachute placed inside the airframe

Thin plastic sheeting on all four sides contains the unmanned ground vehicle and prevents it from snagging on anything upon deployment. Thin plastic sheeting was used for its durability and light weight. This sheeting also prevents the unmanned ground vehicle from shifting during the launch sequence or during flight.

5 Justification

The bay door design eliminates the potential for the unmanned ground vehicle to snag on anything while exiting the plane. It also opens reliably, closes quickly, and prevents unintended openings.

6 Conclusion

The bay door design will allow us to successfully drop the unmanned ground vehicle and successfully complete of the remainder of the mission.

Brigham Young University

AUVSI Capstone Team (Team 45)

Parachute Folding

ID	Rev.	Date	Description	Author	Checked By
GV-007	1.0	2-1-2019	Created	Jacob Willis	Brandon McBride

Introduction

The parachute is a critical part of the UGV drop system. Ensuring that it unfolds in a repeatable, consistent manner will allow better prediction of where the UGV will land after deployment. This will allow us to meet our airdrop accuracy key success measure.

Definitions

Gore A single fabric piece of the parachute. The alternating red and white pieces.

Shroud The strings attached to the bottom of the gores.

Spill Hole The hole in the top of the parachute.

Swivel The metal coupling at the end of the shrouds.

Parachute Folding Steps

A video of the parachute folding process is found in the capstone box folder at: https://byu.box.com/s/l2p32ai9ylidmniiqcs3lx7ycjyawz1b

1. Shake out the parachute and untangle the shrouds until you can separate the two bundles of shrouds. Make sure the shrouds are as untangled as possible.
2. Pull the shrouds together by starting at the swivel and sliding your hand along the shrouds until you hit the gores. This will pull the gores together so all the shrouds are at the same tension.
3. Organize the gores to have two even sets of six, one on the right and one on the left. Make sure to pull all of the gores that are folded into the middle out so they lay flat. Also make sure to keep one hand, or a twist tie, holding the shrouds together as you do this.
4. Lay the parachute on the table, shrouds in the middle and six gores on each side.
5. Flatten out the parachute and make everything as flat and even as possible. This is subjective, but make sure there aren't any major wrinkles in the chute.
6. Using a 5 inch cardboard rectangle, fold the parachute starting from the bottom. There should be four folds, and the folds should end up in a "Z" shape.

7. Straighten out the shrouds again, making sure they are in a tight, but not tangled bundle at the bottom of the parachute.
8. Fold the shrouds 3 inches up the folded parachute, then bend them to the side and pull them out straight. It is important to make sure the shrouds don't go over the top of the chute, as this will result in a failure to open.
9. Roll the chute around the shrouds, keeping the fabric of the chute as uniform as possible.
10. Continue rolling the shrouds around the chute, trying to keep each line as flat as possible (don't bunch them up) and spiraling down towards the bottom of the chute.
11. When 6 inches of shroud are left, stop rolling the shrouds.
12. Keeping the chute tight, slide it into the parachute deployment sleeve, keeping in mind the arrow indicating the out direction.

Conclusion

Once the parachute is folded and in its deployment sleeve, it is ready to be installed in the airframe.

Brigham Young University

AUVSI Capstone Team (Team 45)

Parachute Deployment Drop Simulation

ID	Rev.	Date	Description	Author	Checked By
GV-008	1.0	2-20-2019	Created	Jacob Willis	Derek Knowles

Introduction

To better understand the dynamics of the payload drop with a parachute, a simulation was written based on the dynamics described in 'Dropping an Object on a Target' supplement to *Small Unmanned Aerial Vehicles* by Randal W. Beard. The supplement is found at http://uavbook.byu.edu/lib/exe/fetch.php?media=shared:object_drop.pdf. To more closely simulate the changing dynamics of a parachute opening, the surface area and coefficient of drag were made functions of time, which change between a small value (before parachute opening) and a larger value after the parachute opens. These values were determined from typical values and from t Cd reported by the parachute manufacturer. Surface area is estimated based on the UGV and folded/furled parachute surface areas. Other parameters, such as mass, were determined from estimated UGV system values.

Results

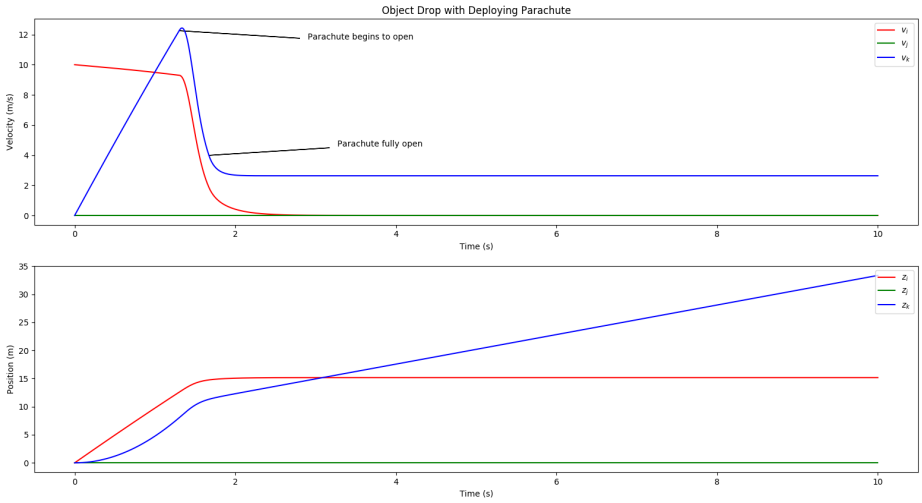


Figure 1: Parachute deployment simulation plots

The main product of this simulation is the plot in Fig 1. Several important results can be seen in this figure. First, the dropped object essentially maintains it original forward velocity until the parachute opens. The parachute's drag occurs in the direction of the object's velocity, so the parachute very quickly reduces horizontal velocities to 0. Another

important result is that the parachute reduces the terminal velocity of the dropped object very quickly, slowing it to a speed low enough for a soft landing.

This informs us of the importance of repeatedly timing the different portions of payload drop sequence in order to accurately predict where our UGV will land.

Simulation Code

```
#! /usr/bin/env python3
"""
    drop.py
    Simulation of parachute unraveling and opening during a UGV drop.
    Author: Jacob Willis
    Date: 29-Jan-2019
"""

import numpy as np
import matplotlib.pyplot as plt


# when the parachute is opening
# times estimated from drop testing with actual parachute
paraTrans = (1.3,1.67) # seconds

# calculate coefficient of drag for opening the parachute after a certain
# time. Piecewise step with a slope between the min and max
def Cd(t):
    minCd = .5 # starting coefficient of drag
    maxCd = 1.5 # final coefficient of drag

    if (t < paraTrans[0]):  # parachute not deployed
        return minCd
    elif (t < paraTrans[1]): # parachute deploying
        return minCd + (t-paraTrans[0])*(maxCd-minCd)/(paraTrans[1] - paraTrans[0])
    else:
        return maxCd # parachute fully deployed

# calculate the parachute area for opening the parachute after a certain
# time. Piecewise step with a slope between the min and max
def Area(t):
    minArea = np.pi*.05**2 # area of closed parachute
    maxArea = np.pi*.5**2 # area of open parachute

    if (t < paraTrans[0]):  # parachute not deployed
        return minArea
    elif (t < paraTrans[1]): # parachute deploying
```

```
        return minArea + (t-paraTrans[0])*(maxArea-minArea)/(paraTrans[1] - paraTrans[0])
    else:
        return maxArea # parachute fully deployed

g = 9.8 # acceleration due to gravity
mass = 450 # grams
rho = 1.2 # density of air

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

t_start = 0
t_end = 10
Ts = .01
tvec = np.linspace(t_start, t_end, (t_end-t_start)/Ts)
v0 = np.array([10, 0, 0]) # initial velocity conditions (i, j, k)
v = np.zeros([3, len(tvec)])
v[:,0] = v0
z = np.zeros([3, len(tvec)])

# run the simulation
step = 0
while step < len(tvec)-1:
  t = tvec[step]
  vs = v[:, [step]]
  v[:, [step+1]] = vs + Ts*(g*k - (rho*Area(t)*Cd(t)*(np.linalg.norm(vs)*vs)))
  z[:, [step+1]] = z[:, [step]] + Ts*vs
  step += 1


# plot the velocity results
plt.subplot(2, 1, 1)
plt.title("Object Drop with Deploying Parachute")
plt.plot(tvec, v[0,:], 'r', label='$v_i$')
plt.plot(tvec, v[1, :], 'g', label='$v_j$')
plt.plot(tvec,  v[2, :], 'b', label='$v_k$')
plt.legend(loc=1)
plt.xlabel("Time (s)")
plt.ylabel("Velocity (m/s)")

# draw arrow for parachute deployment
arrow_x = paraTrans[0]
arrow_y = v[2, int(paraTrans[0]*len(tvec)/(t_end - t_start))]
arrow_dx = 1.5
arrow_dy = -.5
plt.arrow(arrow_x, arrow_y, arrow_dx, arrow_dy)
text_x = arrow_x + arrow_dx + .1
text_y = arrow_y + arrow_dy - .1
plt.text(text_x, text_y, "Parachute begins to open")
```

```
# draw arrow for parachute fully open
arrow_x = paraTrans[1]
arrow_y = v[2, int(paraTrans[1]*len(tvec)/(t_end - t_start))]
arrow_dx = 1.5
arrow_dy = .5
plt.arrow(arrow_x, arrow_y, arrow_dx, arrow_dy)
text_x = arrow_x + arrow_dx + .1
text_y = arrow_y + arrow_dy + .1
plt.text(text_x, text_y, "Parachute fully open")

# plot position results
plt.subplot(2, 1, 2)
plt.plot(tvec, z[0,:], 'r', label='$z_i$')
plt.plot(tvec, z[1, :], 'g',label='$z_j$')
plt.plot(tvec, z[2, :], 'b', label='$z_k$')
plt.xlabel("Time (s)")
plt.ylabel("Position (m)")
plt.legend(loc=1)

plt.show(block=False)
input("Press any key to continue...")
```

Conclusion

The parachute deployment drop simulation successfully allowed us to simulate the drop of our unmanned ground vehicle. Simulation results led us to pick a parachute diameter of 36 inches in order to meet the system requirements for dropping velocity.

Brigham Young University

AUVSI Capstone Team (Team 45)

UGV Drop Test Description

ID	Rev.	Date	Description	Author	Checked By
GV-008	1.0	2-19-2019	Created	Brandon McBride	Jacob Willis

Introduction

This artifact details the methods and results from testing the UGV drop system on the aircraft.

Test Procedures

We first started testing the UGV drop with a parachute and a simulated UGV (water bottle with added weight) from the 2nd floor of the Engineering building to the first to see how gently the parachute will cause the UGV to land. In this test, we already had the parachute deployed and spread out. Next, we tested dropping the simulated UGV from the top of a 50 foot parking garage. This test showed us that the parachute will deploy in time and reach its terminal velocity. Once we tested the dropping of the UGV, we began testing the deployment of the UGV from the aircraft. When we had the bay door built, we set the aircraft on a tall ladder and activated the release mechanism to make sure the parachute wouldn't get caught on the aircraft. After many successful attempts of deploying the UGV from the aircraft in a controlled environment, we dropped the simulated UGV during a flight test.

UGV Drop Test Result

A video of the drop test is found in the capstone box folder at: https://byu.box.com/s/wy5dv7jbyjcpto1uvw0xjjnbu4kw6uwq The UGV was dropped 120 feet from the ground, about the height we would drop it in the competition. The aircraft was flying at 15 meters per second. As soon as the UGV was deployed, the dynamics of the aircraft were marginally altered as seen from the slight dip of the aircraft. The UGV successfully deployed and the landing was gentle. There was a slight drift in the landing as seen from the trail of snow in figure 1 below.

Conclusion

Our incremental testing of the UGV drop lead to a successful deployment in a flight test. Our next step is to integrate the UGV drop with the autopilot system.

Figure 1: The snow trail shows movement of the UGV on impact

Brigham Young University

AUVSI Capstone Team (Team 45)

Unmanned Ground Vehicle Requirements Matrix

ID	Rev.	Date	Description	Author	Checked By
RM-001	0.1	10-23-2018	Initial requirements	Jacob Willis	Brady Moon
RM-001	1.1	10-26-2018	Better performance measures	Jacob Willis	Kameron Eves
RM-001	1.2	10-26-2018	Edits after design review	Brady Moon & John Akagi	Kameron Eves
RM-001	2.0	02-20-2019	Added measured values	Derek Knowles	Brandon McBride

Product: UAV
Subsystem: PAYLOAD/Unmanned Ground Vehicle (UGV)

	Target Design Requirements	Importance	1	2	3	4	5	6	7	Market Responce
	Subsystem Performance Measures		Drop mechanism mass	Weight mechanism can support	Aircraft internal volume consumed*	Stowed drop mechanism drag	Maximum landing velocity	UGV landing distance from target	Rule violations	
	Units		kg	N	cubic meters	N	m/s	m	cnt	
1	Complies with competition rules	5	●						●	Good
2	Capable of lowering the payload to the ground	5	●	●						Very Good
3	Lands UGV within landing zone	3					●	●		Neutral
5	Delivers UGV without damage	3		●			●		●	Good
6	Deployable from airframe	4			●	●				Very Good
7	Does not interfere with takeoff/landing	3	●			●				Very Good
8	Causes minimal aerodynamic interference	3				●				Good
9	Drop mechanism does not interfere with UGV movement	2					●	●		Very Good
	Lower Acceptable		0	0.6	-	0	0	-	-	
	Ideal		0.1	1.3	0	0.3	1	0	0	
	Upper Acceptable		0.6	-	50	1.5	5	22	1	
	Predicted		0.018	3.36	0.003	0.1	15	10	0	
	Measured		0.016	3.43	0.003	N/A*	10	N/A*	0	

*To be measured during system refinement stage

Figure 1: Requirements matrix for the subsystem which will deliver the UGV to the ground.

Brigham Young University

AUVSI Capstone Team (Team 45)

Unmanned Ground Vehicle Drop Subsystem Description

ID	Rev.	Date	Description	Author	Checked By
GV-005	1.0	10-30-2018	Wrote concept description	Kameron Eves	Andrew Torgesen
GV-005	1.1	2-21-2019	Updated to reflect subsystem engineering	Jacob Willis	Brandon McBride

Introduction

This document gives a more detailed description of the selected concept for the UGV delivery system. The concept selected during concept development was a parachute with fins. After considering the added complexity of fins, and the small benefit they provide, we determined to simply use a parachute.

Description

The UGV is to be loaded within the aircraft. 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 system enough to allow the UGV to survive impact without damage. A visual depiction of our chosen system can be seen in Fig. 1.

An accurate landing is an important part of the competition, and is the key success measure governing the design of the UGV drop system. A hole in the top of the parachute improves the horizontal accuracy of the system by allowing a faster drop, but with more consistent aerodynamic effects. This hole is known in the industry as a spill hole because it allows the air to spill out of the center of the parachute. While this will not be enough to correct for large errors, it should be enough to ensure the system doesn't drift too randomly.

The algorithm for dropping objects from a UGV, as detailed in *Small Unmanned Aircraft: Theory and Practice* by Randy Beard and Tim McLain, is used to estimate the proper location to drop the UGV from in order to hit the target. This algorithm uses the wind and velocity of the aircraft to predict the best location to release the payload. A simulation of the drop, based on this algorithm is in GV-008. The parachute adds some complexity to the algorithm, since the surface area and coefficient of drag are not constant during the drop. In order to more accurately predict the landing location of the UGV, the drop will be repeatedly timed, and the time will be used by the algorithm. This will allow us better accuracy in our drop algorithm than using a model of how the parachute drops. It also necessitates a highly repeatable deployment. These were design goals, and were focused on in the development of GV-006 and GV-007.

Figure 1: Our parachute and simulated UGV as seen from the side.

Conclusion

Using the system described above, we are confidant in our ability to achieve a landing accuracy of within 25 feet. This is considered excellent performance in our key success measures and will give us 75% of the points possible in this portion of the competition.

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 competiton. 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

testNextAndPrevRawImage(*interface*)

testNextAndPrevCroppedImage(*interface*)

testCropPost(*interface*, *imgId*)

2.2 Variables

Name	Description
__package__	**Value:** `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=boolean)

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=boolean)

---

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=boolean)

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
    dlong = (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 coordniates 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 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.

		Units	Images/second	Meters/second	Pixels/in	Percent	Count	Count	Percentage	Hertz	Count	
	Product: UAV Subsystem: IMAGING/INTEROPERABILITY **REVISION HISTORY** 1.0: Inital Release	**Subsystem Performance Measures**	Rate of images received on ground	Plane speed	Image resolution	Amount of frame with image	Distinguished colors	Distinguished shapes	Image overlap	Rate of telemetry recieved from plane	Distinguished alphanumeric	
	Target Design Requirements	**Importance**	1	2	3	4	5	6	7	8	9	**Market Response**
1	Detect minimum object size manually	9	■	■	■	■			■			Good
2	Determine object geolocation within 30 m manually	9	■		■					■		Good
3	Detect minimum object size autonomously	9	■	■	■	■			■			Neutral
4	Determine object geolocation within 30 m autonomously	9	■		■					■		Good
5	Rate of ground station contact is 10 hz (Interop)	6	■							■		Very Good
6	Differentiate between five distinct shapes	6			■			■				Very Good
7	Differentiate between five distinct colors	6			■		■					Very Good
8	Determine alphanumeric characters manually	6			■			■			■	Very Good
9	Determine alphanumeric characters autonomously	6			■			■			■	Neutral
		Importance	9	3	9	3	3	3	3	6	3	
		Lower Acceptable	0.5	7	0.7	2	3	3	5	10	2	
		Ideal	1	12	1.1	30	5	5	20	20	5	
		Upper Acceptable	N/A	15	N/A	90	10	13	N/A	N/A	26	
		Predicted				5	10	13	5		20	
		Measured	0.5	13	1.67	TBD	TBD	TBD	TBD	20	TBD	

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

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 confg.ini file

Overrides: object.__init__

close(*self*)

Safely close the DAO's connection. It is higly 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__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

2.1.2 Properties

Name	Description
Inherited from object	
__class__	

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 confg.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
Inherited from object	
__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 confg.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
Inherited from object	
__class__	

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 confg.ini file

Overrides: object.__init__ extit(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=boolean)

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

5.1.2 Properties

Name	Description
Inherited from object	
__class__	

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 confg.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
Inherited from object	
__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 confg.ini file

Overrides: object.__init__ extit(inherited documentation)

upsertCropped(*self*, *manualCropped*)

Upserts a cropped image record. Will only insert or update values persented 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
Inherited from object	
__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
__package__	**Value:** `None`

9 Module src.dao.model.incoming_gps

9.1 Variables

Name	Description
__package__	**Value:** `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__	**Value:** `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__	**Value:** `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__	**Value:** `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 historicall 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__	**Value:** `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 historicall 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__	**Value:** `'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 confg.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
Inherited from object	
__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 confg.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__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

17.1.2 Properties

Name	Description
Inherited from object	
__class__	

18 Module src.dao.util_dao

18.1 Class UtilDAO

object
src.dao.base_dao.BaseDAO
src.dao.util_dao.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 confg.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
Inherited from object	
__class__	

19 Module src.ros_ingest

19.1 Functions

main()

19.2 Class RosIngester

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	**Value:** `10`

Index

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.

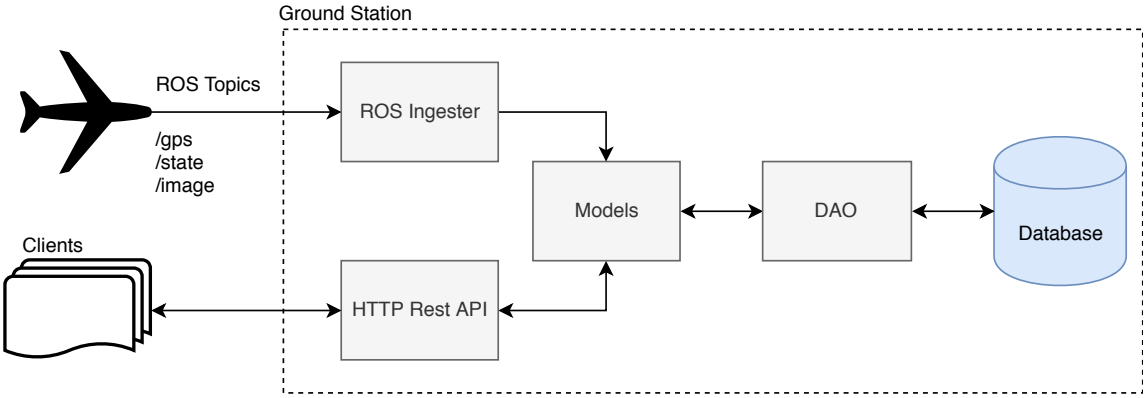


Figure 1: Server Architecture

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.

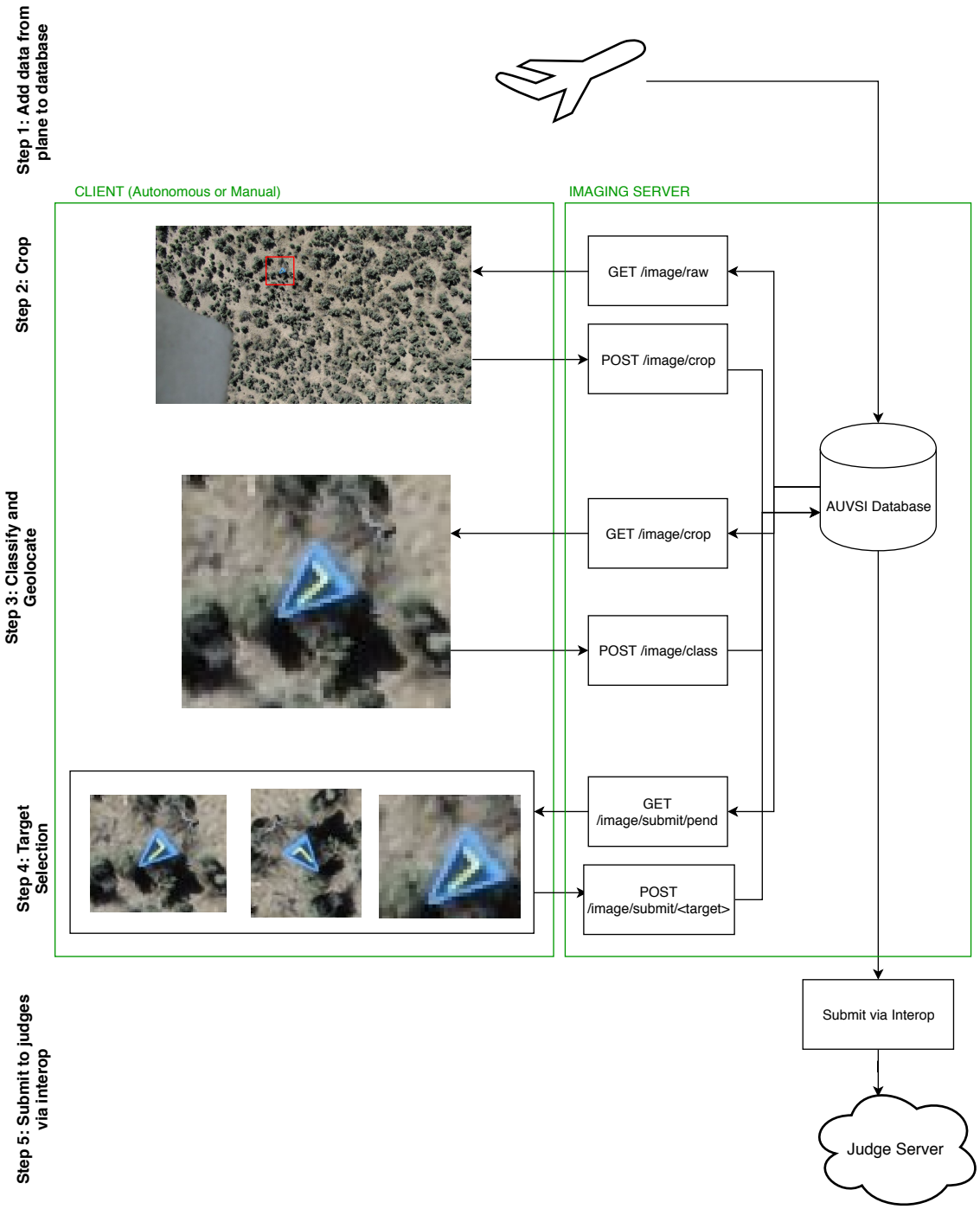


Figure 2: AUVSI Imaging Data Flow Through Autonomous/Manual Classification

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 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 dont 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 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


(1987, 970)

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.