# Situational Awareness Sensor
# Final Report

Prepared for Dr. Sergey Frolov & Patrick Andresen

Prepared by:
Alejandro Caraveo, Krisin Farris, Andrea Golden-Lasher,
Josh Nachtigal,  Zach Smeton

June 18, 2019

## 1.0 Introduction

Sunlight Aerospace  is an aerospace company focused on solar airborne communication. Specifically, Sunlight Aerospace  is working towards the development of High Altitude Long Endurance (HALE) networks. Implementations of these HALE networks by other companies involve the use of a single, large, solar powered aircraft; Sunlight Aerospace  is taking a novel approach by developing a fleet of small unmanned aerial vehicles (UAVs) that work together to achieve the same long endurance goals. By having many aircraft working together in a drag-efficient formation, loss of power can be avoided and the aircraft can stay in the air longer. Because a strict aircraft formation must be maintained to achieve this, the need for systems to identify the relative location of UAVs in the fleet is paramount.

This project's objective was to develop a Situational Awareness Sensor (SAS) to find, identify, and compute the relative location of a leader UAV in a live video feed taken from a follower UAV. Initial requirements for the SAS included running on specific hardware (Raspberry Pi 3 B+ with Pi Camera), achieving a reporting framerate of 20 Hz, and identifying and giving distance estimates of the UAV up to 10 meters away while maintaining distance estimates accurate to within 1-2 cm. These requirements were adjusted later with the help of the client to reflect limitations of the provided hardware and time constraints. Instead, the final requirements of the project became to provide a proof of concept of the requested features by creating a system with the best possible frame rate and best distance estimation accuracies possible on limited hardware; this was done with the knowledge that better hardware could improve the framerate and processing speed of the system in the future.

## 2.0 Requirements

### 2.1 Functional Requirements

The high level functional requirements for this system can be divided into two pieces:

1. A desktop environment where a chosen object detection algorithm can be trained on available UAV video data
2. An executable that runs real time analysis on a UAV video stream and can be updated with new object detection parameters, networking parameters, and different cameras

To accomplish these high level requirements, additional functional requirements are as follows:

- Be able to train and use a chosen object detection algorithm to identify and locate a model SZD-45 OGAR in image
- Use/create identification system to differentiate between members of the swarm
- Use pose estimation methods to find relative position of identified leader aircrafts

- Create an output feed that provides swarm members with associated locations and identification using UDP Protocol
- Transfer information to specific IP address and port
- Identification and detection of objects should hit a target of 70% while in frame [1]
- Identification of distances should hit a target of being within 1 meter of true distance
- An ideal pose estimation reporting rate of 20 Hz or as fast as possible

### 2.2 Non-Functional Requirements

- The solution must be implemented on a Raspberry Pi 3 Model B+
- The live video feed has been obtained from a Raspberry Pi 1080p camera
- File sharing was  done through the encrypted platform Sync
- GitHub was used for version control for code
- Python was used for development

### 2.3 Deliverables

The product was delivered as two separate components. The first components were delivered  as a Raspberry Pi and camera system. This Raspberry Pi includes the software required to identify the leading aircraft and its relative location. Additionally, it implements the ability to transmit this information over UDP and log it to CSV files. The software component of the project was delivered via Github in addition to what is loaded on the Raspberry Pi. This portion of the system includes the neural network, training interface, executable analysis script, and network interface to ensure that all the configuration files are made and trained properly.

## 3.0 System Architecture

The Situational Awareness Sensor can be thought of in two parts: the real time system and the training/configuration/testing system. The real time system is what runs on the Raspberry Pi on the UAVs in the air. The purpose of this part of the system is to detect or track, identify, and estimate the pose of the UAV in the live video feed. Additionally, this part of the system must communicate and log this information. Thus, the real time system is split into five sections: detection, tracking, identification, pose estimation, and communication. Each of these sections is described in detail in the below architecture discussion as well as the Technical Design section.

The training/configuration/testing system is meant to be run on a computer on the ground, before and after in-flight use. It is recommended to run on a computer with a GPU, as this portion of the system is very computationally intensive. This section is also broken up into subsections, described below as well.

The training/configuration/testing system, as shown in Figure 1, was made to create the configuration files for the system, train the neural network used within the system, and run tests on the system. The configuration files were made from a desktop environment and correspond to the camera calibration, identification configuration, the neural network training and the object configuration. The camera calibration is specific to each camera and must be done on an individual basis to ensure that they are all providing undistorted images. If the images are distorted, the radial distortion will cause errors in the bounding boxes of the images and give incorrect estimations on distance. The identification configuration is made using a simple Hue Saturation Value (HSV) masking suite that allows for the specification of values that are allowed to be seen through filtering of the camera feed. These hues are specifically correlated to an identification number assigned to each of the aircrafts. The neural network model, currently working from a darkflow Tiny YOLO version 1 model, was trained using pre-labeled images. The object configuration details in the dimensions of the UAV, for our purposes the model SZD-45 OGAR.

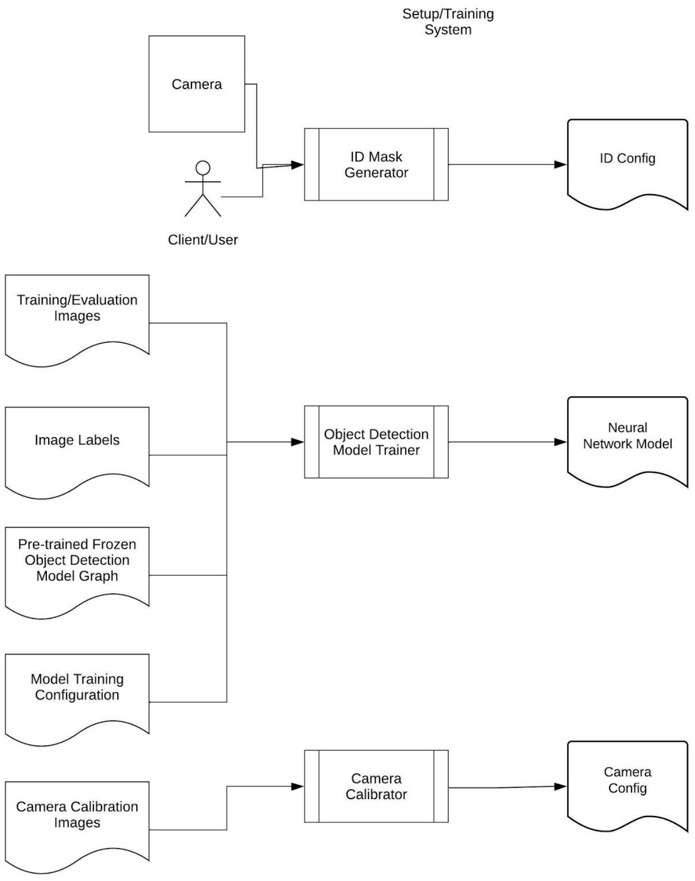

Setup/Training System
Camera
Client/User
ID Mask Generator
ID Config
Training/Evaluation Images
Image Labels
Pre-trained Frozen Object Detection Model Graph
Model Training Configuration
Object Detection Model Trainer
Neural Network Model
Camera Calibration Images
Camera Calibrator
Camera Config

Figure 1: Training/Configuration/ Architecture

The real time system architecture can be seen in Figure 2. This system is made up of many modules and utilizes the configuration files created from within the configuration architecture described above. In this diagram you can see modules that correspond to the different subsections of the real time system: detection, tracking, identification, pose estimation, and communication.

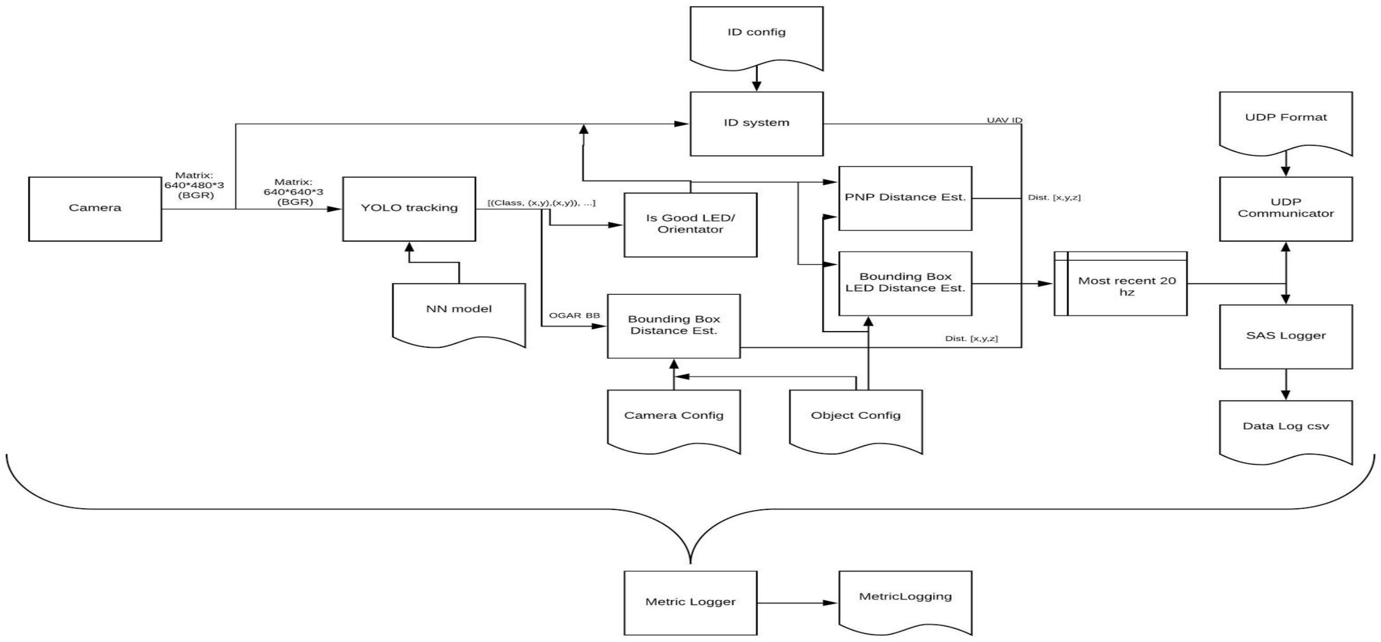


Figure 2: Real Time System Architecture

## 4.0 Technical Design

The following subsections describe details behind the real time and testing/configuration/training portions of the system.

### 4.1 Real Time System

#### *4.1.1 UAV Detection*

The UAV detection system takes in a raw video feed as input and outputs bounding boxes around detected objects within each frame of the video. A graphical example of this output can be seen in Figure 3 below.

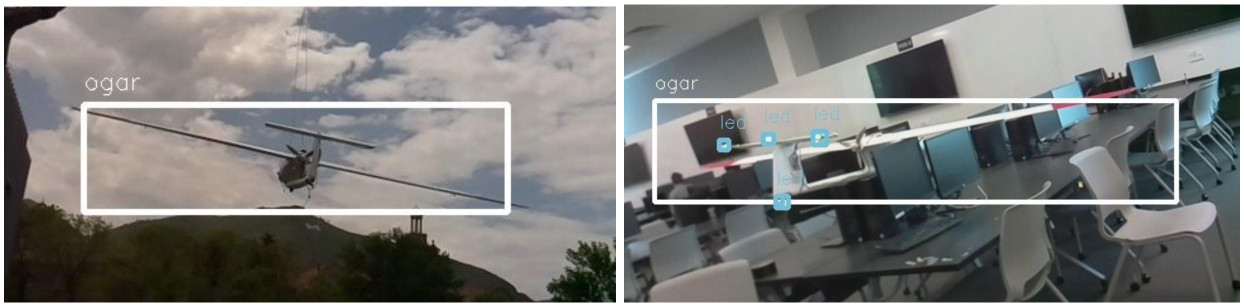


Figure 3: Example output from the object detection

The UAV detection system uses a neural network to identify the UAV model and LEDs that are attached to the aircraft from the live video feed images. To customize the neural network for this project, first the most suitable neural network model for the project's functional requirements was chosen, and then the model was trained on a dataset created by the project team.

The You Only Look Once (YOLO) neural network model was chosen for this project due to its ability to run in real time speed, unlike most other models. Additionally, a lightweight version of YOLO, Tiny YOLO is available, which is significantly faster but slightly less accurate than the full-version YOLO. This project uses Tiny YOLO version 1 for the neural network model.

In order to train a neural network to accurately identify objects, a minimum of 2000 instances of the object in training images are recommended. Training data included 2740 instances of the Ogar model and 8160 instances of LEDs. The neural network framework used to train the Tiny YOLO neural network is Darkflow, which is a Python port of the open source framework Darknet. Training was performed until the average loss--the inconsistency between the predicted and actual values--was at approximately 2.

### *4.1.2 UAV Tracking*

To speed up the system, the computationally intensive object detection was supplemented with object tracking. Object tracking differs from object detection in that it uses an initial position of an object in a frame to follow the object's movement rather than finding an object in an image. As many types of object trackers are purely mathematically based, object tracking can be significantly faster than object detection, which uses a neural network. With the addition of object tracking, the system is approximately 5 times faster.

The OpenCV library provides several different tracker modules for object tracking. Kernelized Correlation Filters (KCF) object tracking looks at direction of change between frames to update locations and has better speed and accuracy than other OpenCV trackers. This is the tracker that was chosen for the SAS system. Each time object detection finds the Ogar or LEDs, KCF trackers are initialized to track the identified objects. Object detection is immediately run again on a separate process while the previously identified objects are tracked until a new detection is available. The usage of the trackers allows the location of the Ogar and LEDs to almost constantly be known and for the slow speed of object detection not to hinder the overall system's performance

### *4.1.3 UAV Identification*

In order to identify which specific fleet member is detected by the object detector/tracker, a system to differentiate between UAVs is necessary. This was done using colored LEDs added to the tail of the UAVs, whose locations are poinpointed during the UAV detection/tracking portion of the system. Cropped images of these LEDs are fed into a color identifier that applies color Hue Saturation Value (HSV) masks to the image. For this application the applied masks are red, green, and yellow; more masks can be created and added using the mask creator discussed in the configuration system section below. The masks created are made up of an array of numbers, either 1 when the color they are looking for is present in the pixel or 0 when not present. Whichever mask contains the most 1s corresponds to the correct ID of the plane, and a numeric value corresponding to that ID is returned.

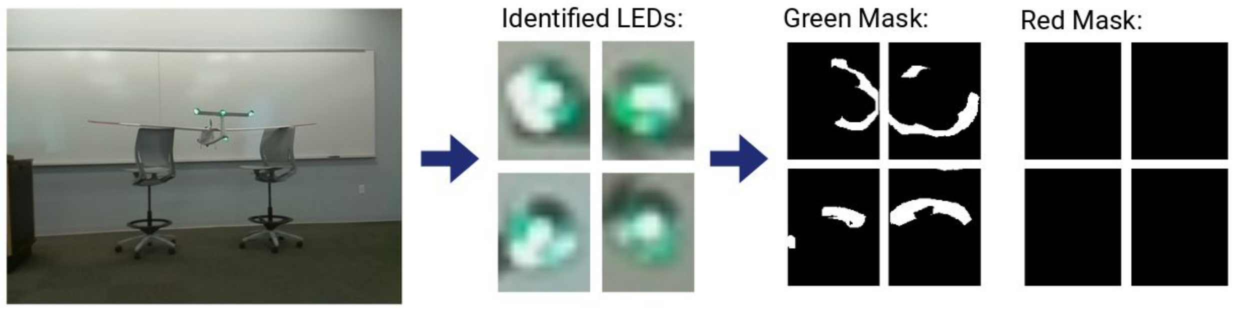


Figure 4: Plane Identification Logic

### *4.1.4 UAV Pose Estimation*

Once the UAV is detected and identified within the fleet, the system must determine an estimation of how far away the UAV is from the camera. Three different calculations are used to estimate this distance.

The first method is a Perspective-N-Point (PNP) calculation using the LEDs identified on the UAV. To do this OpenCV's solvePnP function uses the known distances between 4 or more points, the location of the 4+ points within the image, and the intrinsic properties of the camera to find the relative 6D pose, distance coordinates, and rotation matrix of the object within the world frame. More information on PNP and its application for close formation flight can be found in [2].

The other two methods use the size of the bounding box around the plane returned from the neural network and the distance between rudder LEDs to estimate position. These methods use identical calculations based on known properties of the camera with different reference points located on the UAV. The formula used can be found in appendix section 2

The known length corresponds to the width or height of the UAV in meters for the box estimator. For the point estimator it corresponds to the distance between the left and rightmost rudder LEDs in millimeters. The perceived length corresponds to the length or height of the received bounding box in pixels for the box method. For the point method this corresponds to the distance in points between the left and rightmost LED.

All other variables in this calculation are constants determined from the camera calibration or known values based on the camera used. The focal length is a property of the camera that represents how sharply light converges through the lens. This is calculated for our image size using the formula shown in appendix 7.3.

The image length is the resolution (height or width) of the input image in pixels.
The sensor size is a known property of the camera being used. For example the Raspberry Pi camera used in this case has a sensor size of 3.67 mm [4].

### *4.1.5 Communication and Logging*

Once the UAVs identification within the fleet and relative distance has been found, that data must be sent wirelessly to the flight computer of the UAV. Additionally, that data must be logged locally to CSV files.

For the communication with the flight computer we are creating a peer-to-peer, ad-hoc WiFi network between the flight computer and the SAS. The information is sent as a string over User Datagram Protocol (UDP) to the flight computer. As the Raspberry Pi is not able to process the information at the desired framerate, the sender had to be designed to send the most recent information at 20Hz using shared memory and multiprocessing.

For logging the SAS output data, a csv writer saves the data at the end of the process. Data is only saved at the processing speed of the sensor instead of the 20Hz of the wireless communication. This way only updated data is saved.

## 4.2 Training/Configuration/Testing System

### *4.2.1 Neural Network Training*

After project completion, future users must be able to easily train the neural network on additional training data and different object types to improve object detection performance and expand the scope of the system's functionality. A training system was created to allow users to create, train, and test YOLO neural network models without needing to understand the complexity of the Darkflow framework. This training is intended to be done on a computer on the ground prior to flight, preferably with a GPU as it is very computationally intensive. This training system consists of an intuitive command prompt user interface that guide the user through the process of generating necessary configuration files and settings to properly train a network model

### *4.2.2 Camera Calibration*

In order to accurately be able to calculate distances using a camera, certain camera parameters such as the focal length and a distortion matrix must be calculated. To make this easier for the user, a command line camera calibration tool was created to be run on the real time system's camera before use.

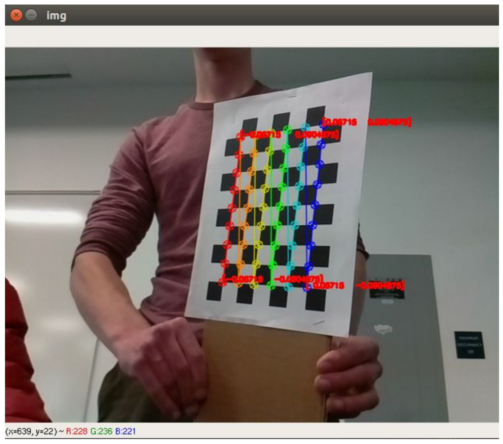


Figure 5: Camera Calibration Tool

### *4.2.3 Testing Suite*

A testing suite containing 18 testing videos was created to gauge the performance of the system. These videos were recorded at known distances away from the leader UAV. At many points during the development of the system, the testing suite was run on these videos to produce average performance and system statistics. A culmination of these statistics were created by graphs, examples of which can be seen below. All of the final graphs created by the testing suite can be seen in the Appendix.

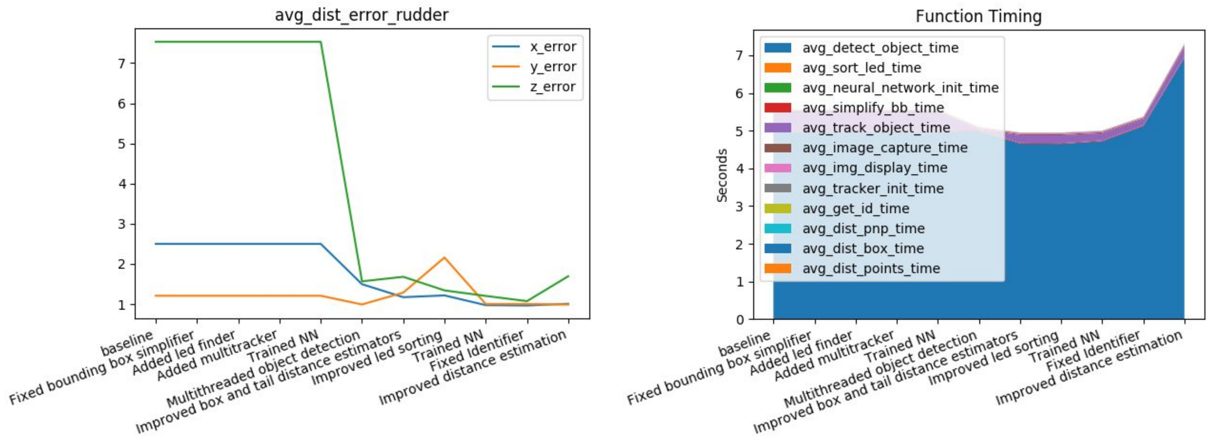


Figure 6: Example testing suite output graphs

## 5.0 Quality Assurance

In order to assure the quality of the Situational Awareness Sensor, many steps have been taken to ensure proper testing practices and coding standards. These steps are summarized in the following sections.

### 5.1 Integration Testing

Integrating the software to ensure compatibility between all the different components in order to produce accurate measurements for the clients use is an increasingly important and difficult thing to accomplish. “Big bang” integration testing was performed to ensure the entire system functions as a whole. Entire system integration was “ground tested” by running the software on the Raspberry Pi in ideal indoor conditions and “flight tested” with a fleet of two aircraft. Integration between each module was tested by ensuring the outputs from one module successfully can be inputted by the subsequent module for various test cases. One example of module integration is testing that the bounding box from the neural network can be successfully transferred to the three distance estimating modules in various testing cases.

### 5.2 User Interface Testing

The software includes user interfaces for neural network training and testing, color mask creation, logging, and camera calibration. Manual testing was done on the different user interfaces used in this software including the tools used for neural network training, color mask creation and camera creation. The graphical interfaces are somewhat simple to test since there is a limited set of actions the user can take in the interface. The command line prompts on the other hand are more thoroughly tested for whatever inputs a user could create.

### 5.3 Code Metrics

To ensure code adheres to the SOLID principles and good programming practices, this software adheres to the PEP-8 style guide. This ensures code is easy to read and extend once the software is delivered to the client. Additionally, using open source code metric analyzes such as Radon helped identify complex code to refactor.

### 5.4 User Acceptance Testing

The user, or client, will use this system in addition to their currently in use systems to accurately gauge distance between UAV aircraft. Distance measurements should be written out to a logging file as well as communicated to other aircraft through UDP communication. The client wants a distance measure to be within a few inches to centimeters. The system will be checked by the client from the ground after the flight has taken place since these drones are meant to fly autonomously.

### 5.5 Performance

The performance of our system is measured in several ways. Each function call is timed and the timing result is written to a log for analysis of what aspects of the program take the most time. Additionally CPU usage, memory usage, and other metrics are logged to ensure the program makes efficient use of hardware resources. The results of the neural network are logged to determine how often it successfully identifies the object. Finally, the distance estimates are compared to actual distance measurements to analyze the overall accuracy of the estimates.

## 6.0 Results

### 6.1 Summary of Testing

The final software system includes a comprehensive testing environment which runs the main program against a set of 18 testing videos. This environment computes the average accuracy of detection, tracking, and distance estimation systems and also performs timing analysis. Graphing all statistical data from the test environment shows improvement of the system as optimizations were performed. This environment also ensures the software is well integrated and runs smoothly without errors. Unit tests of the code were done prior to the compilation of the integrated system to ensure non-working portions were eliminated prior to adding the code into the integrated program. Manual testing was done throughout the development process and was the only testing method ran on GUIs and UIs. Flake 8 and Radon were run on all code to ensure low cyclomatic complexity and adherence to the PEP8 style guide

**6.2 Performance testing results**

The best performance recorded by the testing environment has the following characteristics:

- Average plane detection percentage: 87%
- Average ID accuracy: 80%
- Average FPS (on Raspberry Pi 3 B+): 1.666
- Average FPS (on ASUS with Intel core i7 processor, Ubuntu 18.04): 16.315
- Error in bounding box distance estimation: X: 0.4 m Y: 1.2 m Z: 0.4 m
- Error in PNP distance estimation: X: 1.3 m Y: 1.1 m Z: 0.7 m
- Error in rudder distance estimation X: 0.7 m Y: 1.0 m Z: 1.7 m
- Average CPU Usage (on Raspberry Pi 3 B+): 91.25%

**6.3 Results of Usability Tests**

The completed system was tested during a live flight test during Week 6 to demo the system in an environment that is similar to the high altitude environment that the final system is intended for. Although our current system did not perform as well in the air as it does in our grounded test environment, baseline results and additional data that could be used to improve the system for use in flight were obtained. Based on this final flight test, improvements to the system were made before the final product was delivered to the client.

**6.4 Future Work**

This system was created and trained to recognize a OGAR model airplane; the actual UAVs used by Sunlight Aerospace will likely be different models and may not be detected by the neural network model this project developed. To correct for this, new training data will have to be collected and labeled to allow this system to function for other aircraft. This system could also be extended to recognize other types of objects such as manned aircraft to aid in object collision avoidance. Further training of the neural network on new class types could be used to make this extension. Additionally, user interfaces such as the neural network training environment and camera calibration could be extended to be more user-friendly by including GUIs.

**6.5 Lessons learned**

We learned many things throughout the completion of this project. First, we learned that creating a dedicated testing environment makes performing integration testing much easier and allows improvements in the system over time to be tracked and backed up with data from these integration tests.

Throughout our project there were many times when one group member was stuck trying to debug an issue or find the source of an error in results. Bringing in another person to look over the issue or talk through it almost always solved the issue much faster than it would have been solved alone.

Keeping good records of changes, library versions, and terminal commands makes repeating or recreating the development environment much easier. Many times, errors were caused by a team member using the wrong version of libraries like Tensorflow or OpenCV than what the software was designed to work with. This issue was exacerbated by situations when many libraries relied on each other and on incompatible versions of libraries. Not keeping good documentation on the steps and libraries required to install and run a component of the software, such as the neural network, also resulted in wasted time debugging.

We also learned that test driven development was difficult when creating a hardware-based project. In most classes before this, we were given very structured instructions on what features to add and where they would go; when developing this product we didn’t always know what function would be implemented, how they would work, and what hardware they would be utilizing. Manual testing was much easier to do but often came back to hurt the project later without an easy way to find which class/function was causing which bug.

## 7.0 Appendix

### 7.1 Conventions

To ensure that the software is consistent there is a set of standards on the units and coordinate frames. All distance measurements are done in meters.

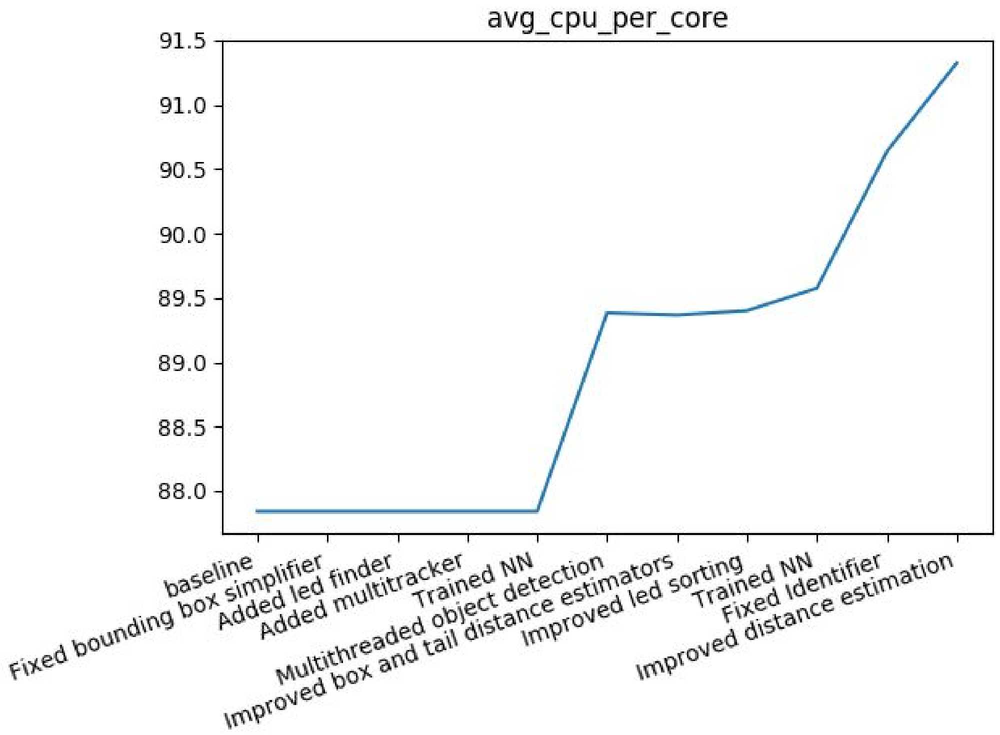
### 7.2 Box and Point Distance Calculations

$$\text{distance from object (mm)} = \frac{focal\ length\ (mm) * known\ length\ (mm) * image\ length\ (px)}{perceived\ length\ (px) * sensor\ size\ (mm)} \quad [3]$$

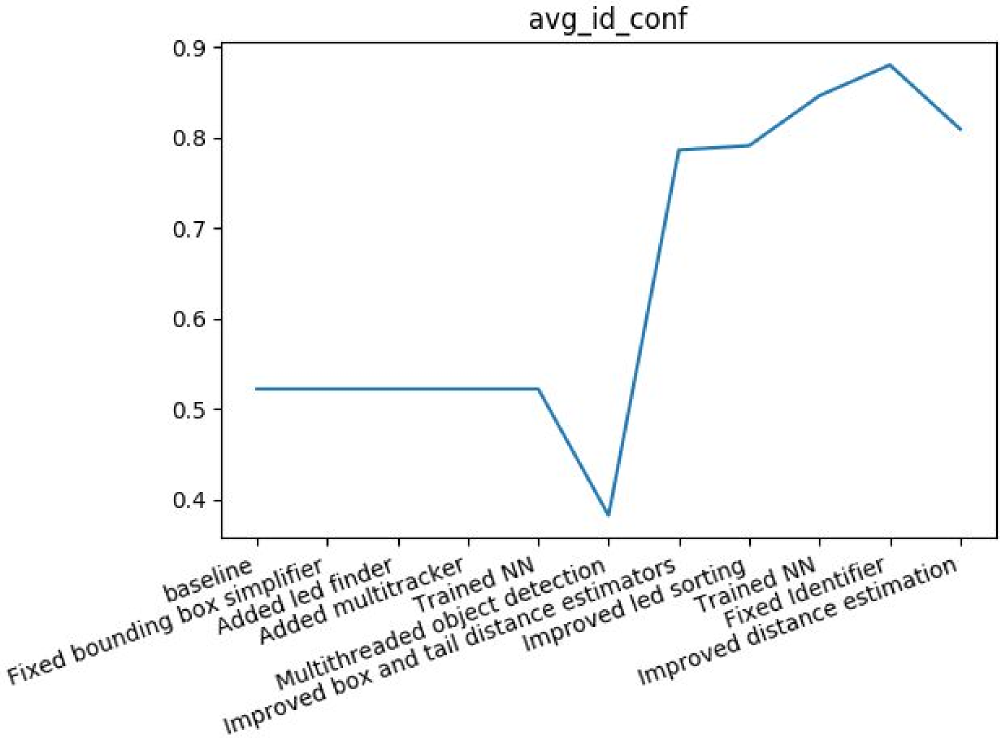
### 7.3 Focal Length Calculations

$$\text{focal length (mm)} = \frac{camera\ matrix\ width\ (px) * sensor\ size\ (mm)}{image\ length\ (px)}$$

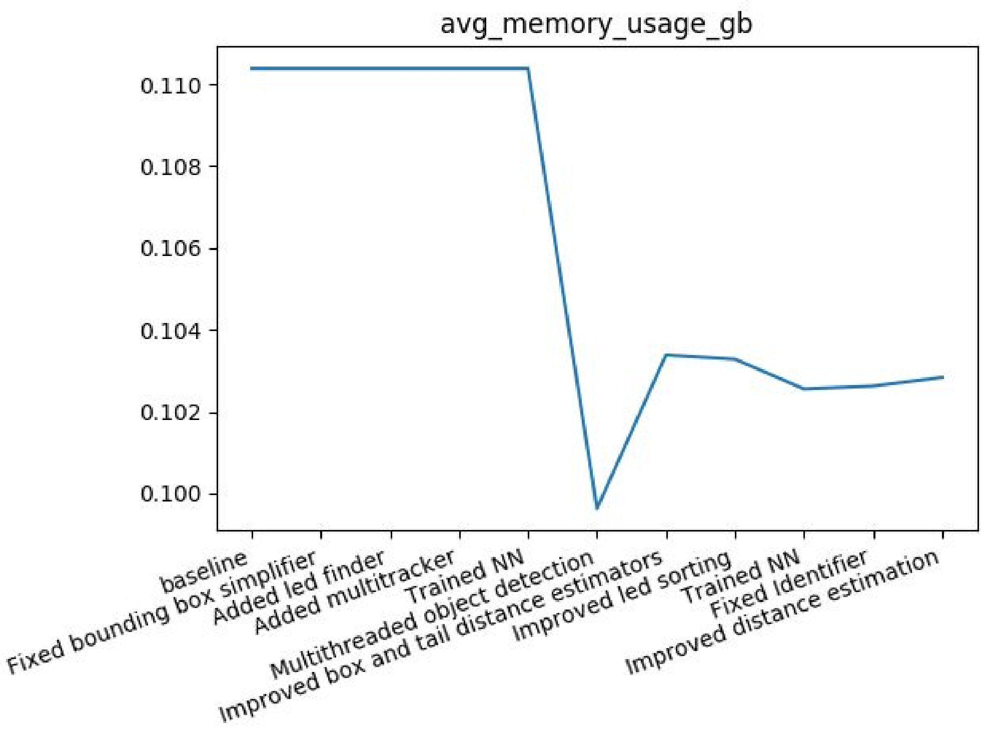
## 7.4 Final Testing Suite Graphs



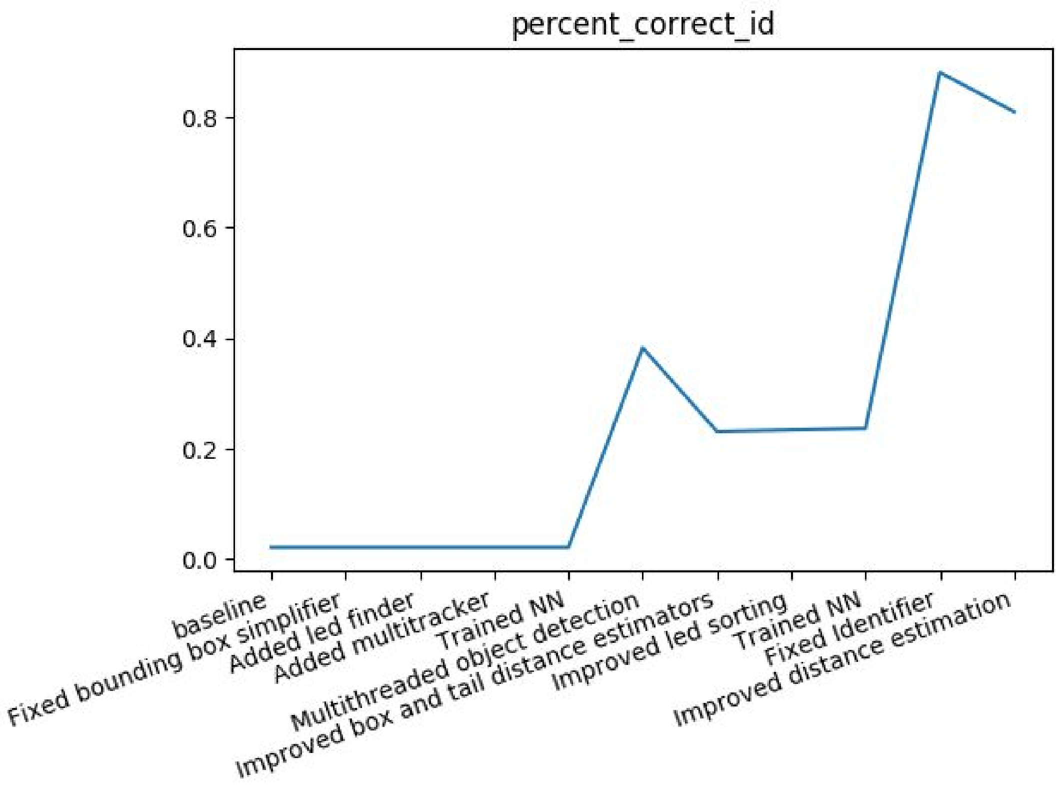
**7.4.01:** Average CPU Usage Per Core on Pi for each System Iteration



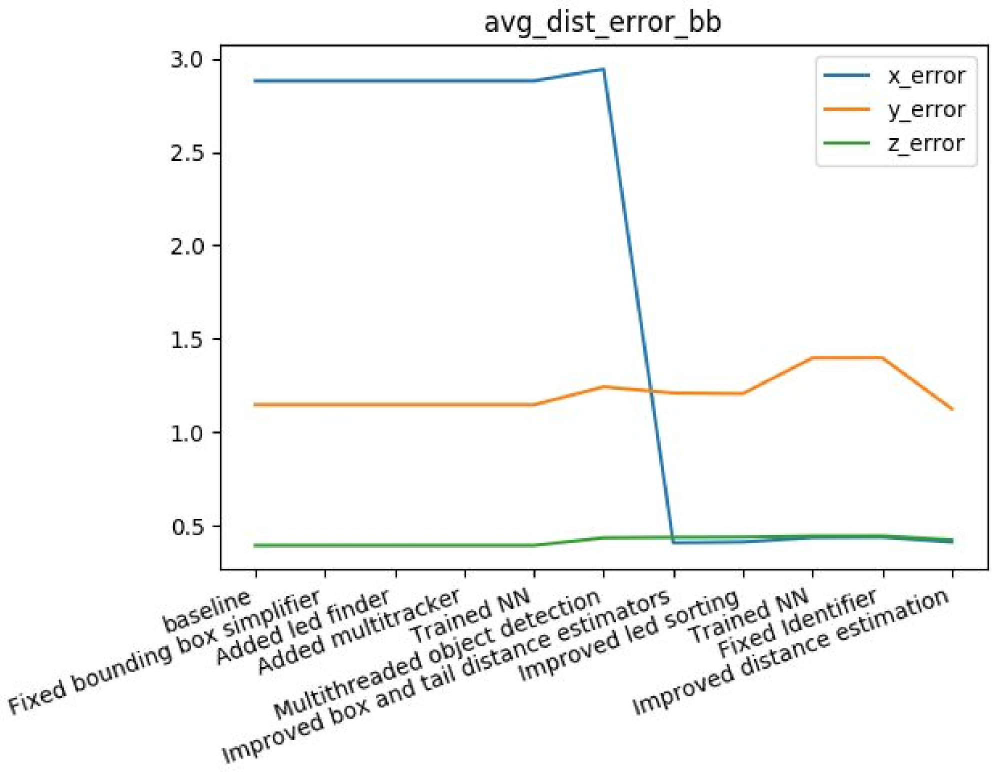
**7.4.02:** Average UAV Identification Confidence for each System Iteration



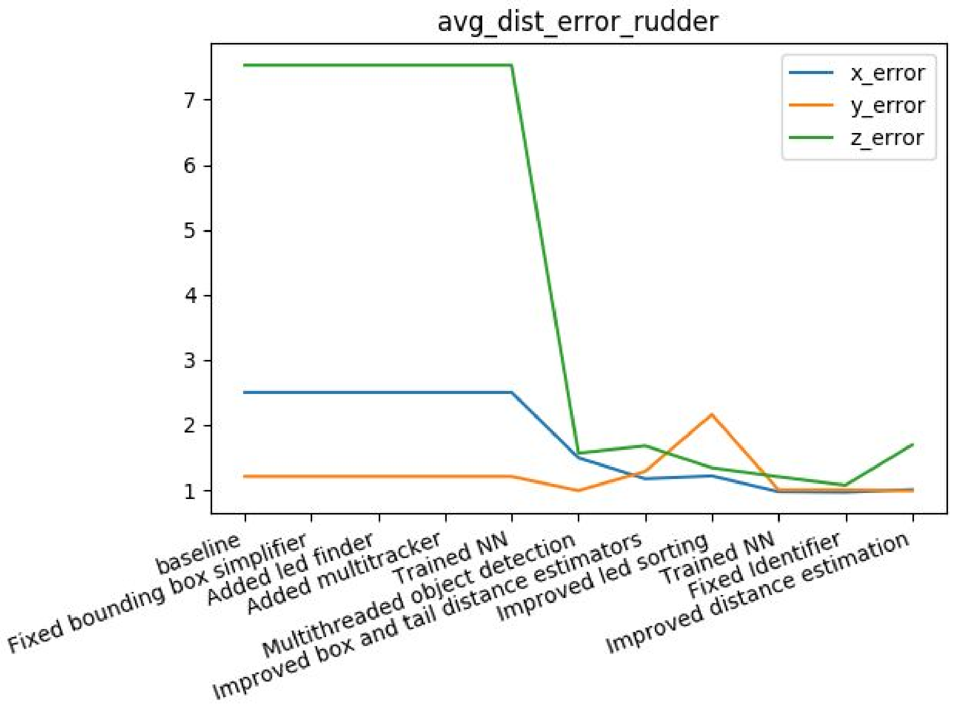
**7.4.03:** Average Memory Usage in GigaBytes for each System Iteration



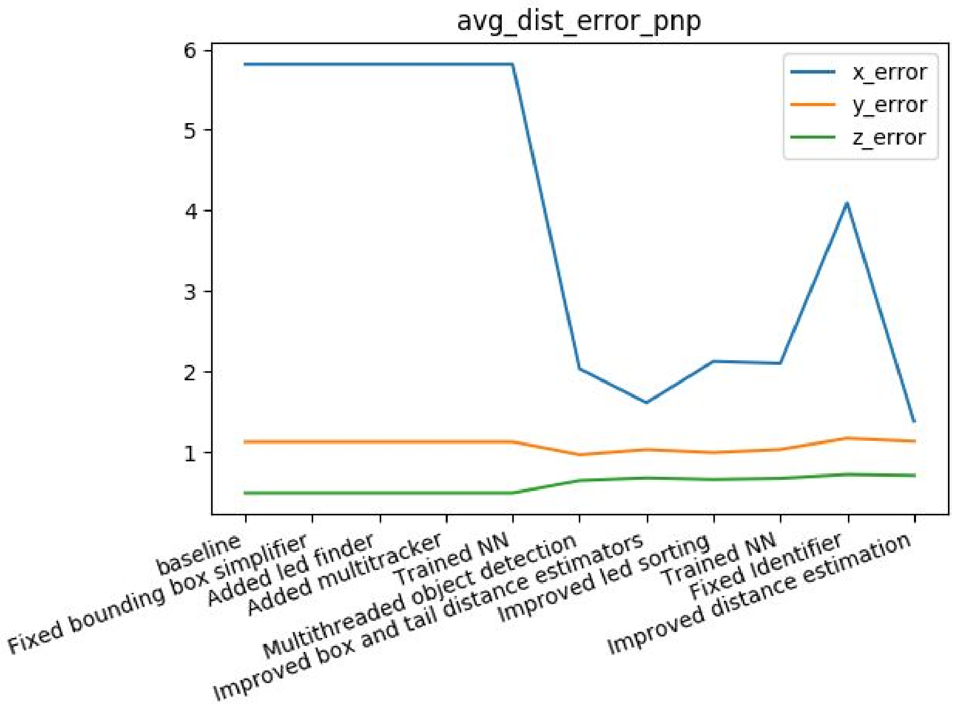
**7.4.04:** Percent of Correct Identifications for each System Iteration



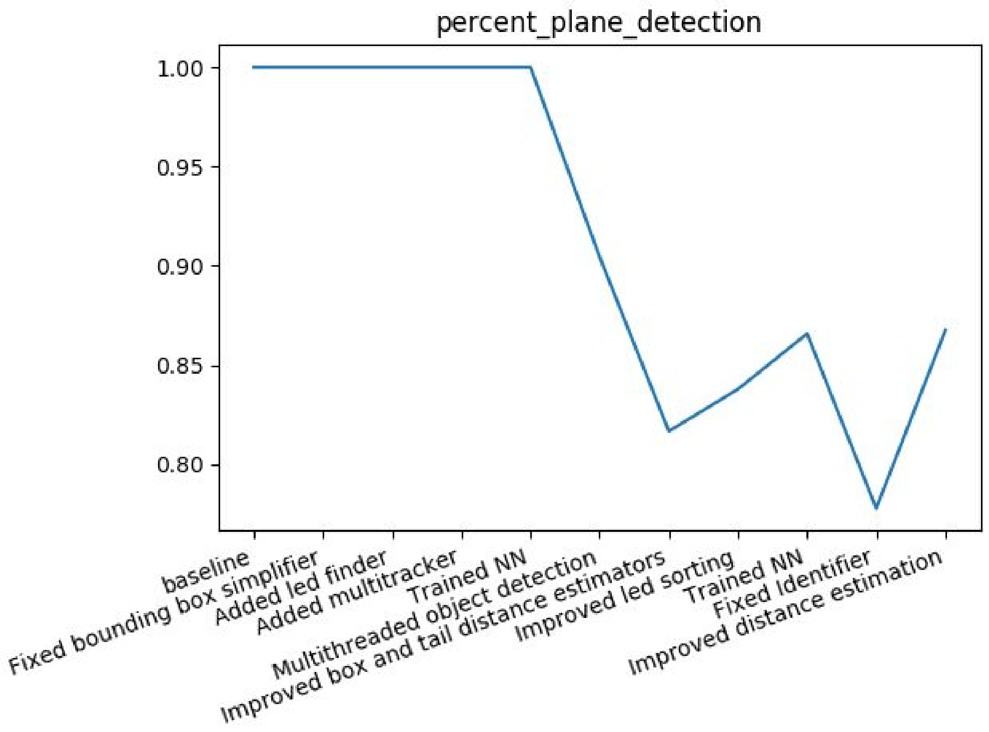
**7.4.05:** Average Bounding Box Estimation Error for each System Iteration



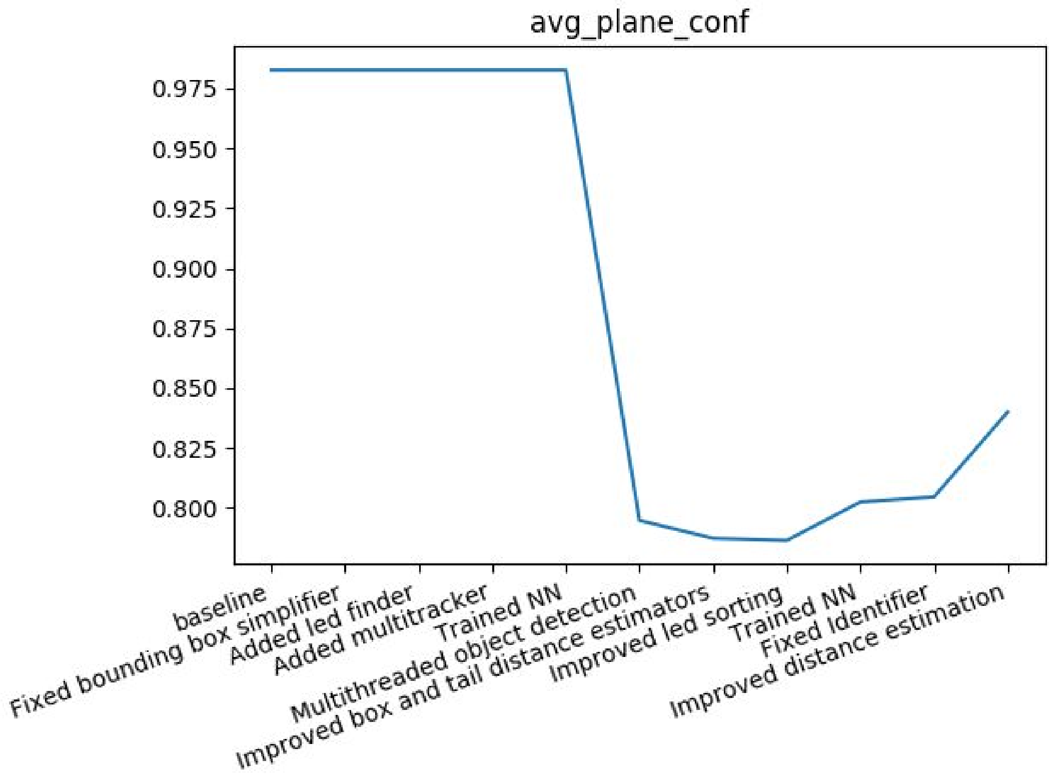
**7.4.06:** Average Rudder Estimation Error for each System Iteration



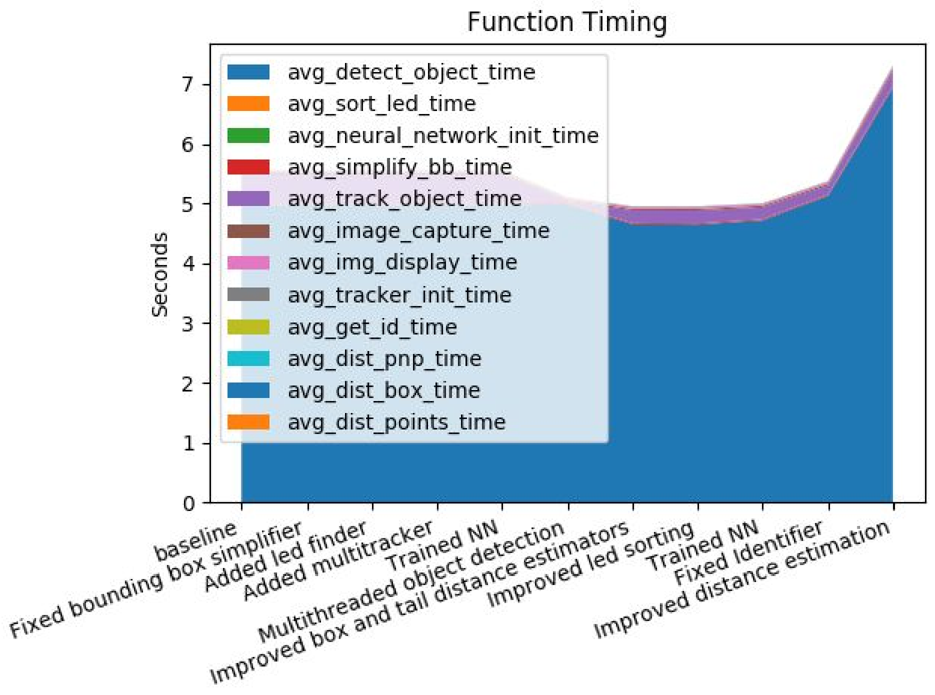
**7.4.07:** Average PNP Estimation Error for each System Iteration



**7.4.08:** Percent of Successful UAV Detections for each System Iteration



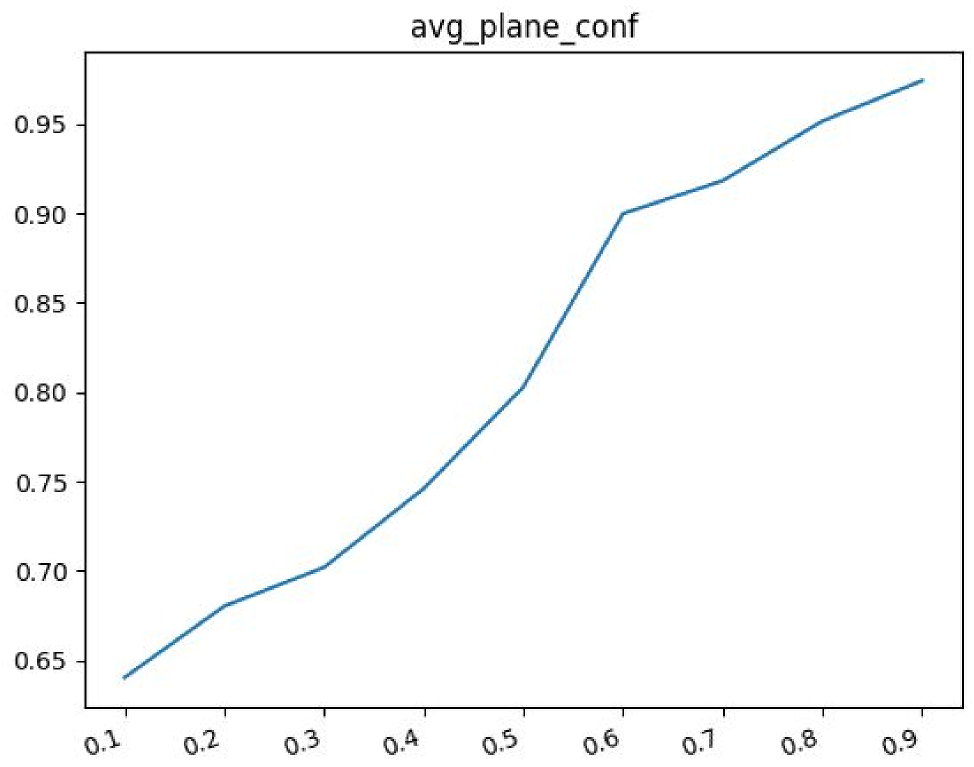
**7.4.09:** Average UAV Identification Confidence for each System Iteration



**7.4.10:** Average Function Runtime for each System Iteration

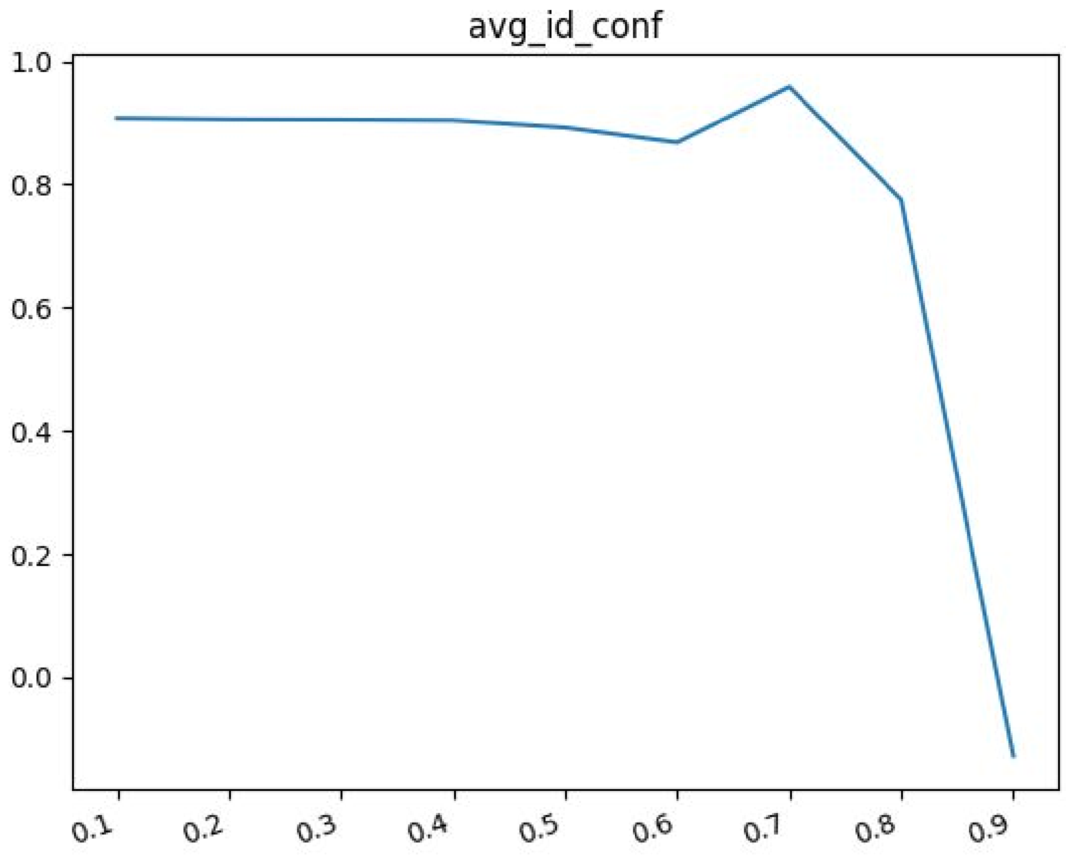
## 7.5 Confidence Changes

Changing the confidence threshold of the neural network determines what level of confidence the neural network must have in its detection in order to return a bounding box. The following graphs illustrate how the confidence threshold affects the outputs of the system. The final system is set to run at a threshold of 0.6

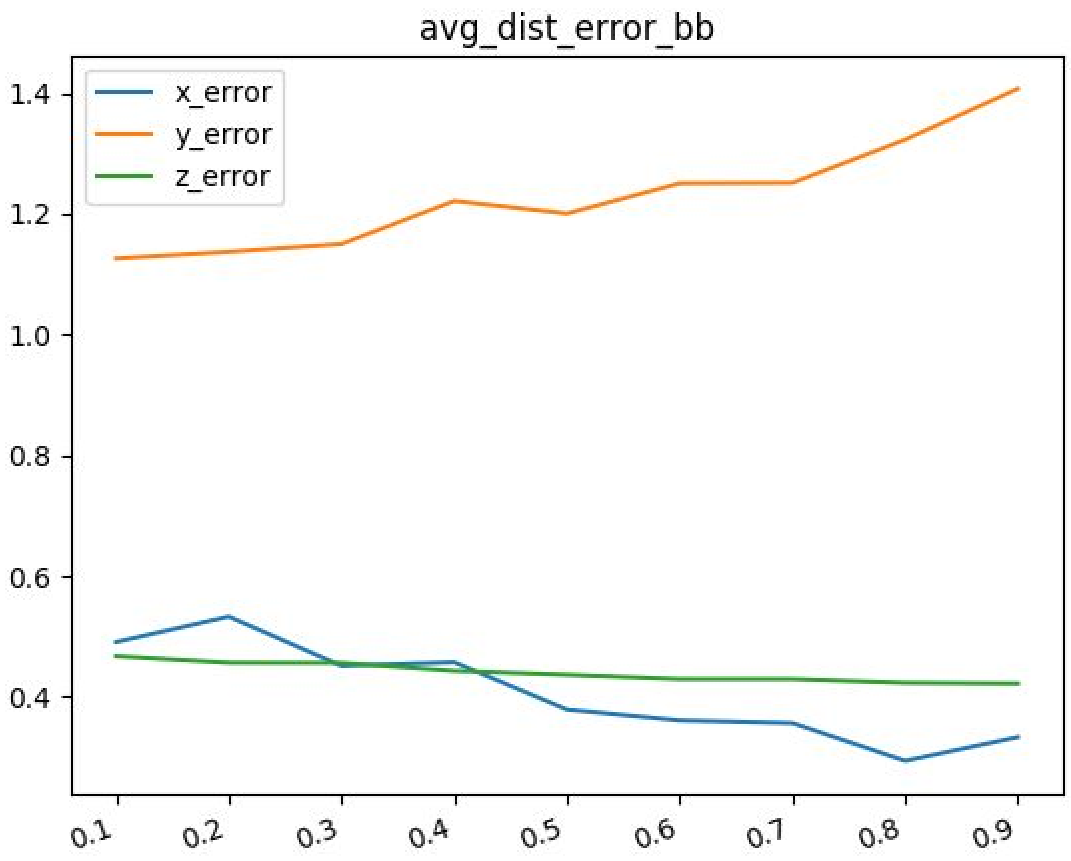


.

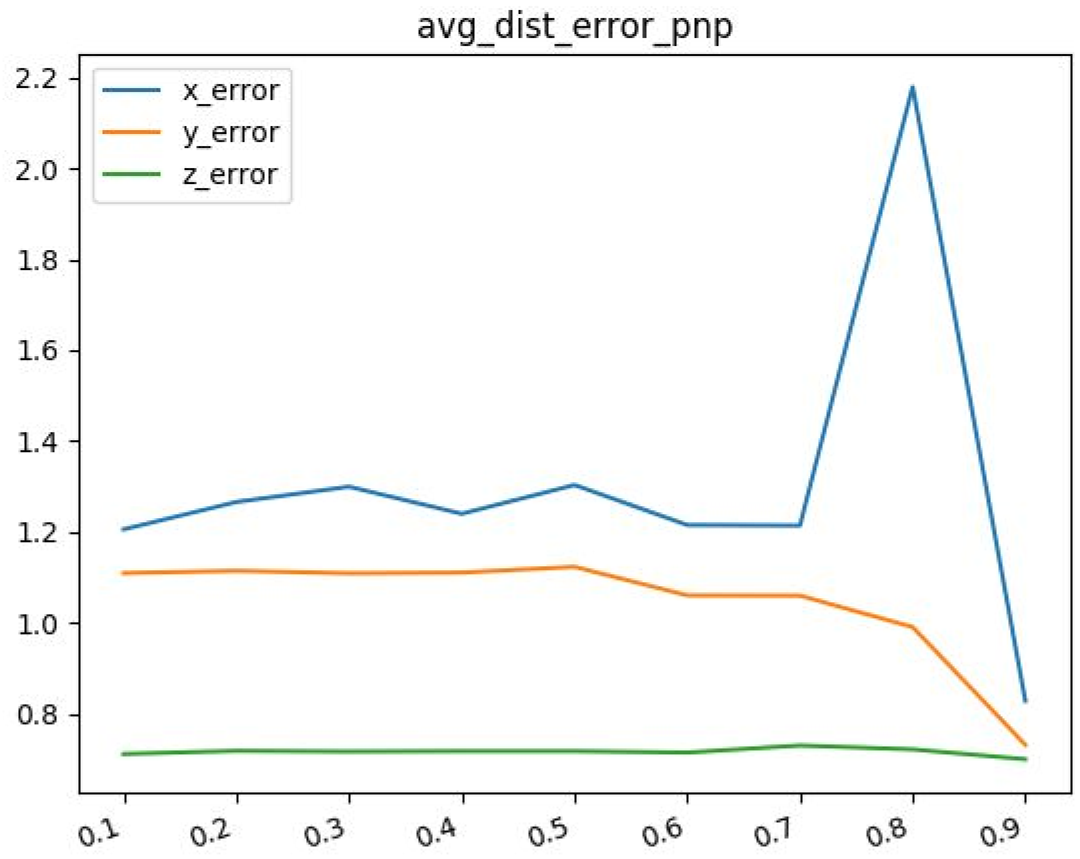
**7.5.1: Plane Detection Confidence**



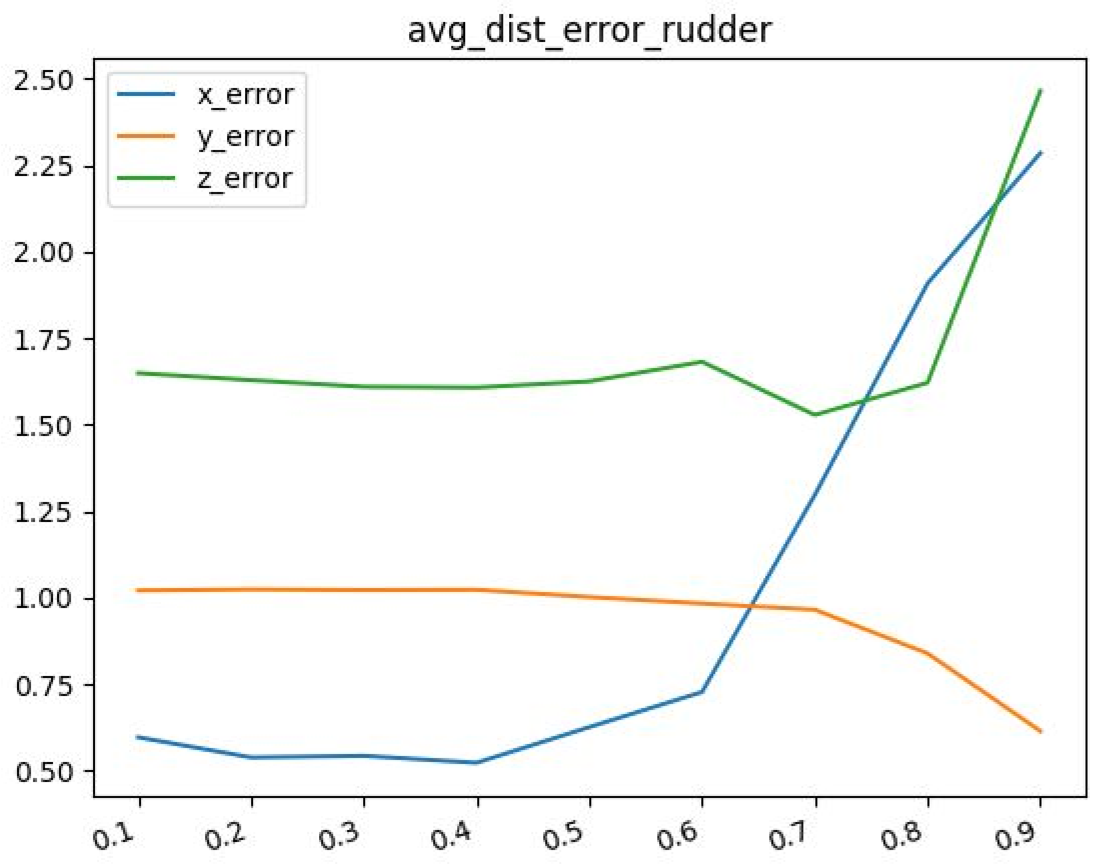
**7.5.2: Plane Identification Confidence**



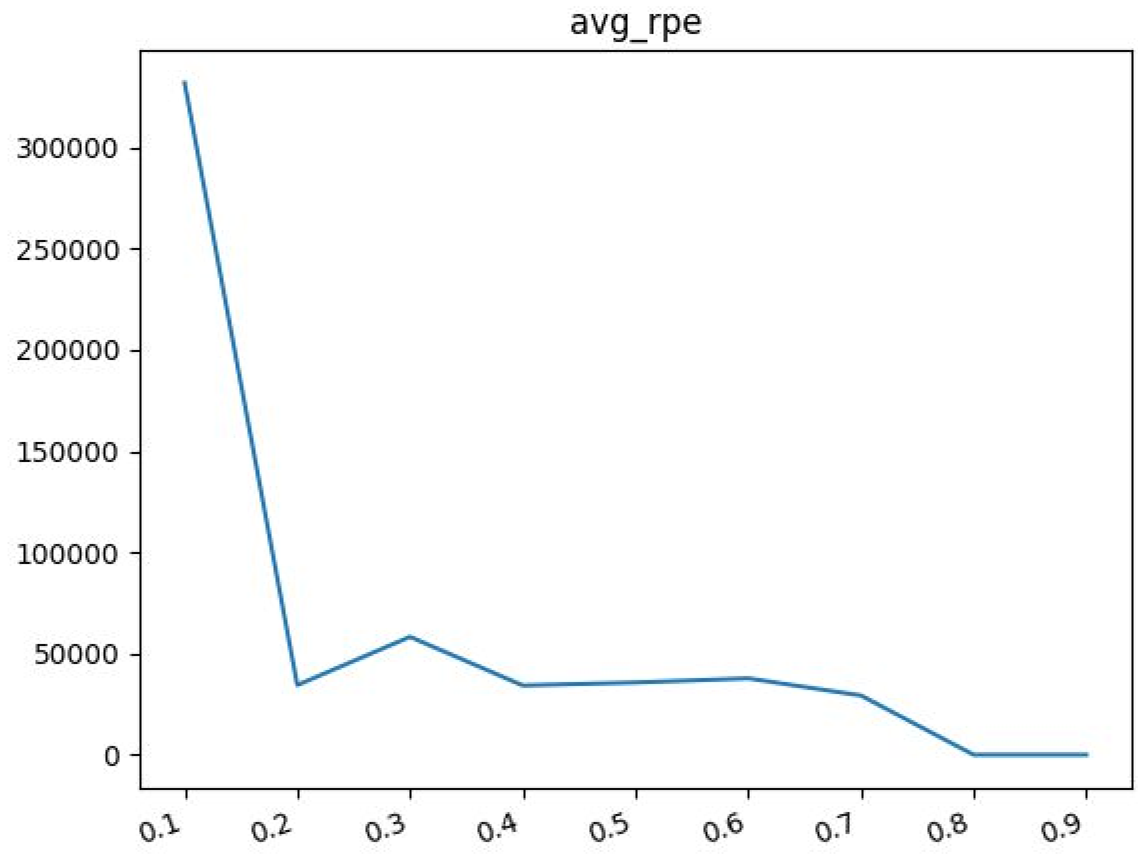
**7.5.3: Errors in Bounding Box Distance Estimation**



**7.5.4: Errors in PNP Distance Estimation**



**7.5.5: Errors in Rudder Distance Estimation**



**7.5.6: Average RPE at Different Confidences**

## 8.0 References

[1] J. Redmon, "YOLO: Real-Time Object Detection", *Pjreddie.com*, 2019. [Online]. Available: https://pjreddie.com/darknet/yolo/. [Accessed: 17- Jun- 2019].
[2] M. Darling, "AUTONOMOUS CLOSE FORMATION FLIGHT OF SMALL UAVS USING VISION-BASED LOCALIZATION", Masters, California Polytechnic State University San Luis Obispo, 2014.
[3] Palmer, C. and Thakur, A. (2019). *OpenCV: How-to calculate distance between camera and object using image?*. [online] Stack Overflow. Available at: https://stackoverflow.com/questions/14038002/opencv-how-to-calculate-distance-between-camera-and-object-using-image [Accessed 17 Jun. 2019].
[4] Raspberrypi.org. (2019). *Camera Module - Raspberry Pi Documentation*. [online] Available at: https://www.raspberrypi.org/documentation/hardware/camera/ [Accessed 17 Jun. 2019].