UNITED STATES MILITARY ACADEMY

FINAL PAPER

ME389: INTRO TO ADVANCED STUDY IN MECHANICAL ENGINEERING

SECTION Z8

MR ANDREW KOPEIKIN

By

CADET BRIAN SAVIDGE ’20, CO C2

WEST POINT, NEW YORK

7 MAY 2019

\_BS\_\_ MY DOCUMENTATION IDENTIFIES ALL SOURCES USED AND ASSISTANCE RECEIVED IN COMPLETING THIS ASSIGNMENT

\_\_\_\_\_ I DID NOT USE ANY SOURCES OR ASSISTANCE REQUIRING DOCUMENTATION IN COMPLETING THIS ASSIGNMENT

SIGNATURE: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Unmanned Aerial Vehicle Swarm: Mapping Algorithms**

CDT Brian Savidge, Mr. Andrew Kopeikin

Corresponding author's Email: brian.savidge@westpoint.edu

**Author Note:** Thank you to Defense Threat Reduction Agency, Mr. Ross Arnold, and Major Dominic Larkin. Special thanks to Mr. Andrew Kopeikin for his guidance and support.

**Abstract:** This study explored the capability of an unmanned aerial vehicle (UAV) swarm to locate and survey areas of interest as quickly as possible. The swarming process involves decentralized control in which UAVs decide their respective paths, as opposed to the use of a centralized node with total control. The implementation of a new swarming algorithm, Savidge Swarm, was found to be capable of conducting such missions much more efficiently than Greedy Goto, a baseline algorithm. This was determined by conducting simulations in SCRIMMAGE, a computer program designed for drone swarm simulations. Savidge Swarm adds communication amongst UAVs and periodical re-planning during the swarming process. This study analyzed how much more effective the new swarming method was than the baseline version.The Savidge Swarm algorithm was additionally live flight tested to prove real-world capability.

*Keywords:* Unmanned Aerial Vehicle, Drone Swarm, Swarming Algorithm

# 1. Introduction

Despite the efforts to prevent the nuclear proliferation, nuclear weapons are still prominent and remain a threat in the modern world (Kristensen, 2019). In the event of a nuclear blast, the primary step after evacuation is to identify the most affected regions and determine the impact location. However, due to the large-scale effects of nuclear weapons, it is both time consuming and dangerous for personnel to investigate these areas. The use of unmanned aerial vehicle (UAV) swarms would allow such areas to be inspected relatively quickly, and, unlike a satellite image, can provide a very detailed analysis of the given area (Cook, 2017). Not only are the swarms efficient but can conduct such missions without putting human lives in danger (Kopeikin, 2019). This project was originally geared towards radiation blasts specifically in support of the Defense Threat Reduction Agency (DTRA) Post Blast Recon Swarm Capstone at West Point, but the swarming algorithms discussed below can also be applied to other scenarios such as search and rescue teams or enemy surveillance.

In using UAV swarms for such tasks, it is important to use the most efficient and effective algorithm possible to gather critical information in as little time as possible. The goal explored in the project is to use a UAV swarm to effectively survey an area and direct the search to primarily focus on suspected areas of interest.

Section 2 of this paper discusses related works and the primary motivations for this research. In Section 3, there is an explanation for the Savidge Swarm algorithm and how it compares to Greedy Goto. Section 4 analyzes the performance of the new algorithm versus the old one, and Section 5 examines a physical implementation of Savidge Swarm. Section 6 summarizes the conclusions drawn from the paper and explains a few methods for the further improvement of Savidge Swarm.

2. Related Works

Swarming algorithms differ across disciplines depending on the end goal, whether it is to locate a source, find hostile enemies, or map a general area. This section intends at looking further into methods of localization and how they can be integrated together for an improved swarming algorithm.

Particle Swarm Optimization (PSO) was one of the first effective source localization algorithms, developed in 1995. This method has undergone several iterations of development since being created and continues to be improved upon (Eberhart & Shi, 2001). PSO was designed to model natural swarms, such as insects or birds searching for food. Each UAV is assigned a random position amongst the sampling area, and is then given a random initial velocity and acceleration, which is updated with every movement the UAV makes. The algorithm assigns the acceleration to be weighted such that the UAVs will be more heavily directed towards areas where more desirable values have already been obtained (Eberhart & Shi, 2001).

Another method uses gradient estimation for source localization. In this method, any given number of drones, equal to or greater than three, is set up in a circular formation. The measured values from each drone are recorded and a weighted average is conducted to determine the gradient of the readings, as witnessed in Figure 1. The drones then move in the direction of the gradient and repeat this process (Cook, 2017). Thus, the drones should continue moving closer to the largest value, or the source. This method encounters issues when the drone formation is very far from the source, because the readings are nearly insignificant, or very close to the source, because these readings tend to be inconsistent (Cook, 2017).

.

**Figure 1. UAS readings at different locations to find the direction of the source**

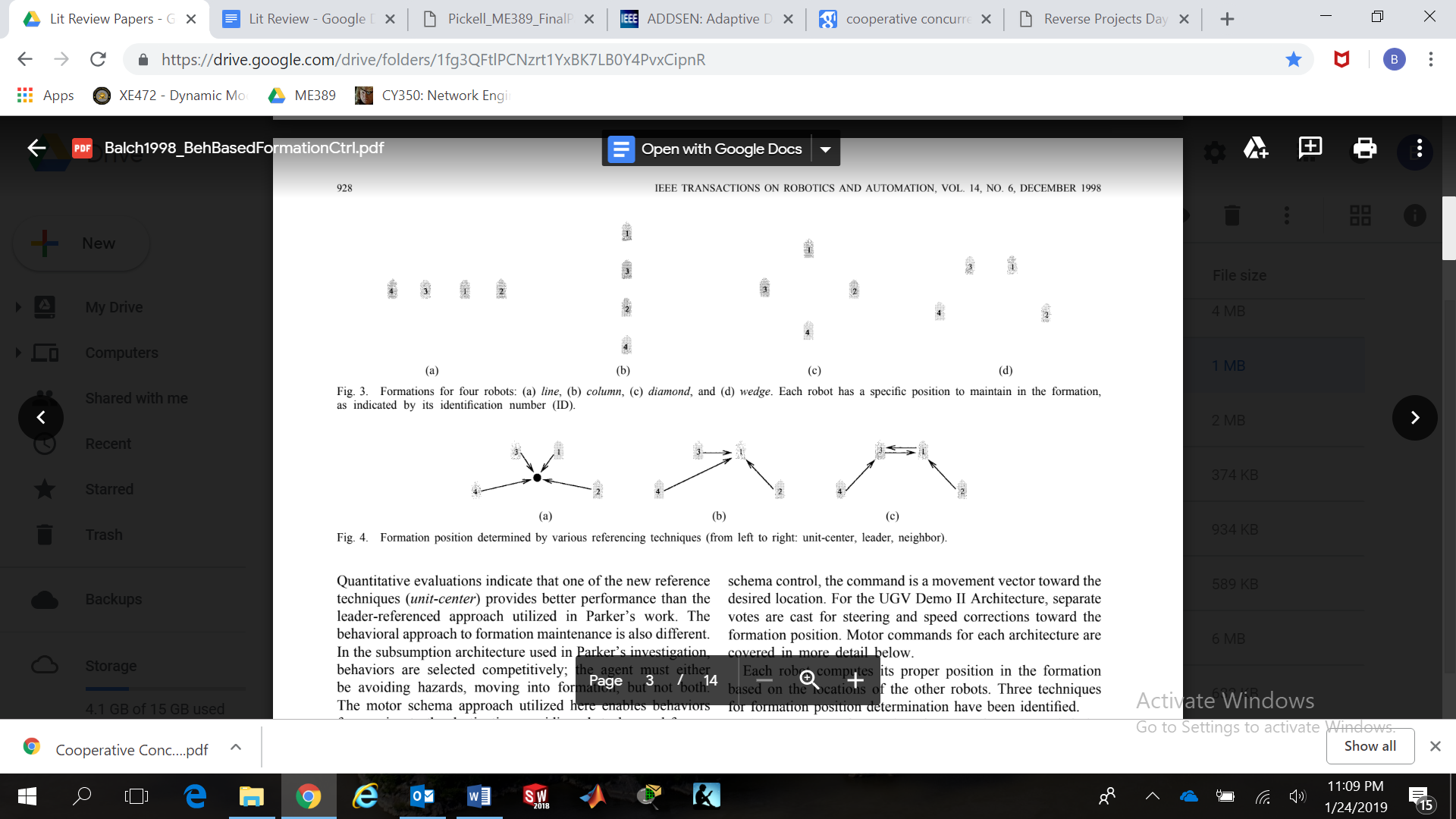
The Parallel Path Planning Algorithm involves a lawnmower-like pattern. UAVs travel relatively close together and clear out one column at a time, before moving to the next pattern (McCune & Madey, 2013). This method covers overlap and is more often used for moving or spreading areas of interest but can also be useful in scanning stationary areas. However, Figure 2 also shows how this method can omit the very edges of a map in order to make it more efficient and prevent excessive overlap between UAVs.



**Figure 2. One iteration of the Parallel Path Algorithm**

The SWEEP Protocol was developed to clear an area without overlapping a single point. The members of the swarm ensure they don’t map any point that would disconnect the graph, as this would force them to cross back over an already mapped area in order to get to the other disconnected portion after finishing with one (McCune & Madey, 2013). This method is limited in that it requires the UAVs to start from different parts of the perimeter, and requires a lot of computing power. However, it can be applied to essentially any desired mapping shape (McCune & Madey, 2013).

In swarming algorithms, there are generally three types of behavior based formation control: unit-center, leader, and neighbor (Balch and Arkin, 1998). Unit-center makes use of a central point that all drones maintain a relative position to. This method was found to be the most reliable. In a leader based formation, one drone is selected to make movements, and the other drones all maintain a relative position to it in order to maintain the formation. The neighbor method is the least reliable as it involves the most variables because each drone keeps a relative position to the two drones nearest itself (Balch and Arkin, 1998), as seen in Figure 3.



**Figure 3. a) Unit-center, b) Leader-based, c) Neighbor method.**

Another swarming method is to let each drone “bid” on their next point. Of the drones that bid on a given point, the closest drone wins whereas any others are left to make a new bid (Sheng, 2006). However, this means there needs to be a bid time for the drone to wait to see if that point is better fit for another drone. This wait time can’t be too short, or else no other UAVs will be given a chance to bid on it, but also can’t be too long or else the exploration time will take far too long. In addition, this method takes into consideration the concept of nearness. The UAVs have limited communication ranges, and thus when choosing points need to remain within a certain distance of other drones, or else they will no longer have the shared data of the swarm (Sheng, 2006). This is especially important when scanning larger areas.

The next point to analyze can also be chosen using a cost-benefit analysis. By determined equations for the cost of moving to each point and the utility of determining values at each point, the most valuable movement can be determined (Burgard, 2000). The cost for moving to each point can be normalized around the distance to each point. The utility of each point becomes more complicated, as the utility is unknown until the data is actually read at that location. Thus, the utility is, by default, set to one, and then utilities are constantly updated to be the average of those around it (Burgard, 2000). This allows the drones to determine which points may be more useful versus not, but requires a constantly updating system. This plot becomes known as the expected information gain; the more information that can be gained in a smaller amount of time, the better (Simmons, 2000).

Among these methods used for drone swarms and localization, a few can be selected and focused upon for swarm efficiency improvement. Particularly, the use of creating a radiation reading gradient to determine the direction of the source, and the use of a cost-benefit analysis to determine the value of travelling to a given location are applicable to the goals of this project. Section 3 further discusses how these concepts were implemented into the swarming algorithm.

3. Swarm Behavior Design

**3.1 Baseline Behavior**

A previously used behavior for locating areas of interest, Greedy Goto, divides maps into waypoints, and each UAV selects the same number of waypoints from this list to visit. The process is only conducted before the swarming begins. It starts with the UAV with the lowest identification number (ID) choosing the closest unassigned waypoint to its current position, then the next point closest to each previously chosen waypoint, until it reaches its share of waypoints. This process is repeated by each subsequent UAV based on ID, resulting in each waypoint being assigned to a UAV. Each UAV individually has to go through this process, predicting what the other UAVs will do in order to determine what their own flight path needs to be. The purpose of Greedy Goto is to map an area of suspected radiation and determine where the radiation hotspots are located. It is used as the baseline in the tests conducted below because it is the previously used method for the DTRA Post Blast Recon Swarm capstone project.

**3.2 Design Modifications**

The new algorithm, refSerred to as Savidge Swarm, makes use of two particular design modifications. First, communication was added to the UAVs such that at each waypoint a UAV takes a reading, it sends a message to all others in the swarm. This message contains the waypoint identifier and the score reading for that waypoint. This *score* could be radiation readings in the case of post blast missions, debris in the case of search and rescue operations, movement in the case of enemy surveillance, or anything else the operator is interested in mapping. Since all UAVs obtain this information, they can begin to predict the location of areas of interest. Second, UAVs update behavior periodically throughout the swarming process. By using newly learned scores to concentrate towards the areas of interest and focus mapping those scores before proceeding to less significant locations.

**3.3 Overall Design Description**

Initially, the structure of the Savidge Swarm method is very similar to Greedy Goto. Before any UAVs initiate movement, the UAVs are assigned initial waypoints in the same manner as Greedy Goto, where the lowest UAV ID takes priority. Now instead of the UAVs being left to collect data and simply send it back to the ground station for compilation, the data gets sent to and stored in all other UAVs in the swarm. A *prediction* matrix is created to maintain what the predicted score is at each waypoint. Each score in the prediction matrix is set to a *default* value, which is the lowest known score. Thus, this default score can change during the course of the search, and it will update to every waypoint in the prediction matrix that has not yet been read, nor predicted based off of a known waypoint’s reading. Essentially, every waypoint has either been surveyed, is predicted based off of a surveyed waypoint, or is predicted as the default score.

After a set amount of time, the UAVs will re-plan their paths according to a new method. Each UAV first individually takes all of the known scores in the swarm. These known scores are used to update the prediction matrix. A variable is set to propagate estimated decrease from the known waypoint to those surrounding it. For example, if this propagation variable is set to 2, and a waypoint was recorded with a score of 4, the propagation would occur as follows: the predicted scores for the 8 waypoints directly surrounding the known score would be , or 2. The predicted scores for the 16 waypoints two away from the initial reading would be , or 1. Thus, the prediction matrix updates using values that have already been read. This process would continue until the propagated score prediction is less than the default score, because the lowest predicted score is intended to be the default value. This process is summarized by Equation 1 below:

, [Equation 1]

where *b* is the predicted score, *k* is the known score, *p* is the propagation value, and n=1, 2, … represents the number of waypoints away from the known waypoint such that the predicted score is greater than the default score. Refer to figure 4 below to see what the belief matrix would like if a reading of 8 was obtained 3 waypoints from the left and 3 from the top. The propagation value is 2 and the default value is 1.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 2 | 2 | 2 | 2 | 2 | *1* | *1* |
| 2 | 4 | 4 | 4 | 2 | *1* | *1* |
| 2 | 4 | **8** | 4 | 2 | *1* | *1* |
| 2 | 4 | 4 | 4 | 2 | *1* | *1* |
| 2 | 2 | 2 | 2 | 2 | *1* | *1* |
| *1* | *1* | *1* | *1* | *1* | *1* | *1* |
| *1* | *1* | *1* | *1* | *1* | *1* | *1* |

**Figure 4. Example of Belief Propagation: the “8” is bold and underlined to indicate it was a read value.**

Note that for clarity and simplicity in this description, a propagation value of 2 was used, but in the actual testing, a propagation value of 1.3 was used. This value was used to allow the belief propagation to spread further, allowing UAVs from further away to be drawn in the direction of higher readings. This is a variable that can easily be modified but was not altered during the tests.

The propagation process is repeated for each newly recorded waypoint since the last update in waypoint distribution among UAVs. The UAV then begins re-planning the path for each UAV in the swarm. Again, the lowest UAV goes first, following a similar process. However, instead of choosing the closest points, the UAVs are now programmed to use a cost-benefit analysis. The benefit, *b*, is measured by the predicted score at a waypoint, while the cost is the distance to travel there. Thus, the value, *v*, for travelling to each waypoint can be calculated according to Equation 2:

[Equation 2]

where *d* is the distance from the UAVs location to a waypoint. This value is calculated for every waypoint in the bundle that has not yet been read. The lowest ID UAV will pick the waypoints with the highest values until it has its share of waypoints that have not yet been read, followed by the subsequent ID UAVs completing the same process (see Appendix B).

Communication is necessary in this process because UAVs need to know what is happening in other locations of the map, not just in their own vicinity. Additionally, communication allows UAVs to predict each other’s paths, so they can appropriately determine their own. Assuming perfect communication, all UAVs will have the same belief matrices at the end of each update period and will all predict the same path for each other. Imperfect communication could lead to discrepancies or even crashes if two UAVs try to read the same waypoint simultaneously due to their inconsistencies.

Algorithm 1 gives a brief summary, in bullet form, of the structure of the swarming algorithm according to the methods described above:

**Algorithm 1:** The *Savidge Swarm* procedure

**input :** *b*: a prediction matrix, *k*: known value matrix,

*t*: time for re-planning process

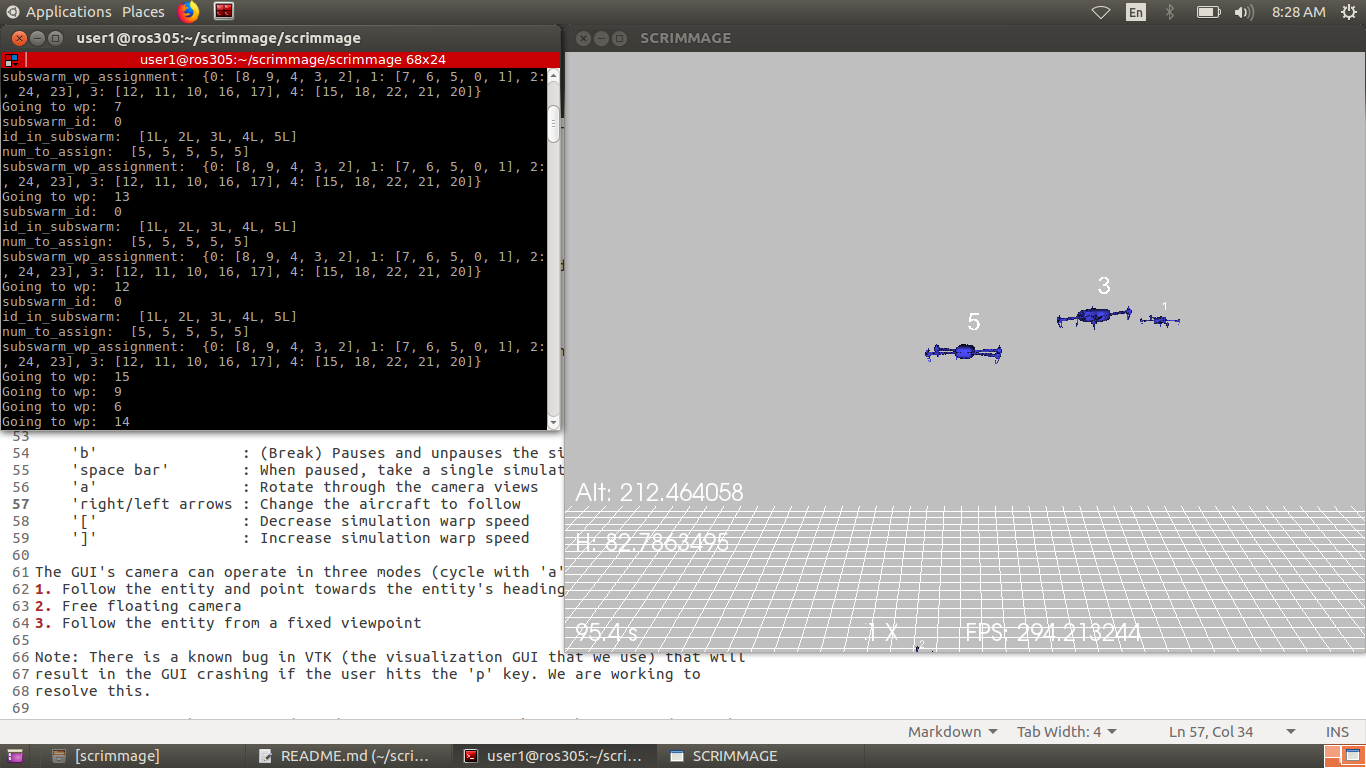
**output :** waypoint path assignment

1. Set all values of *b* to 0 and *k* to blank
2. Initially assign waypoints according to Greedy Goto algorithm
3. After each reading each point, send *k* to all other UAVs
4. After a set time, update *b* based on *k* using equation 1
5. Analyze values for each unassigned waypoint from *b* using equation 2
6. Choose next waypoint as unassigned waypoint with the highest value
7. Each UAV chooses equal number of waypoints
   * Lowest ID UAV takes priority
8. Repeat re-planning after *t*, until no value in *k* is blank

4. Simulation Results and Analysis

**4.1 Data Collection Process**

Greedy Goto was used as a baseline comparison for the performance of the Savidge Swarm algorithm. Data comparisons were made using a simulation program, SCRIMMAGE (“Scrimmage - multi-agent robot simulator”), which allows for rapid development, simultaneous trial runs, and testing the same behavior that can then be implemented on a flight test (see Appendix A). A graphic example of a running SCRIMMAGE simulation can be seen below in Figure 5.

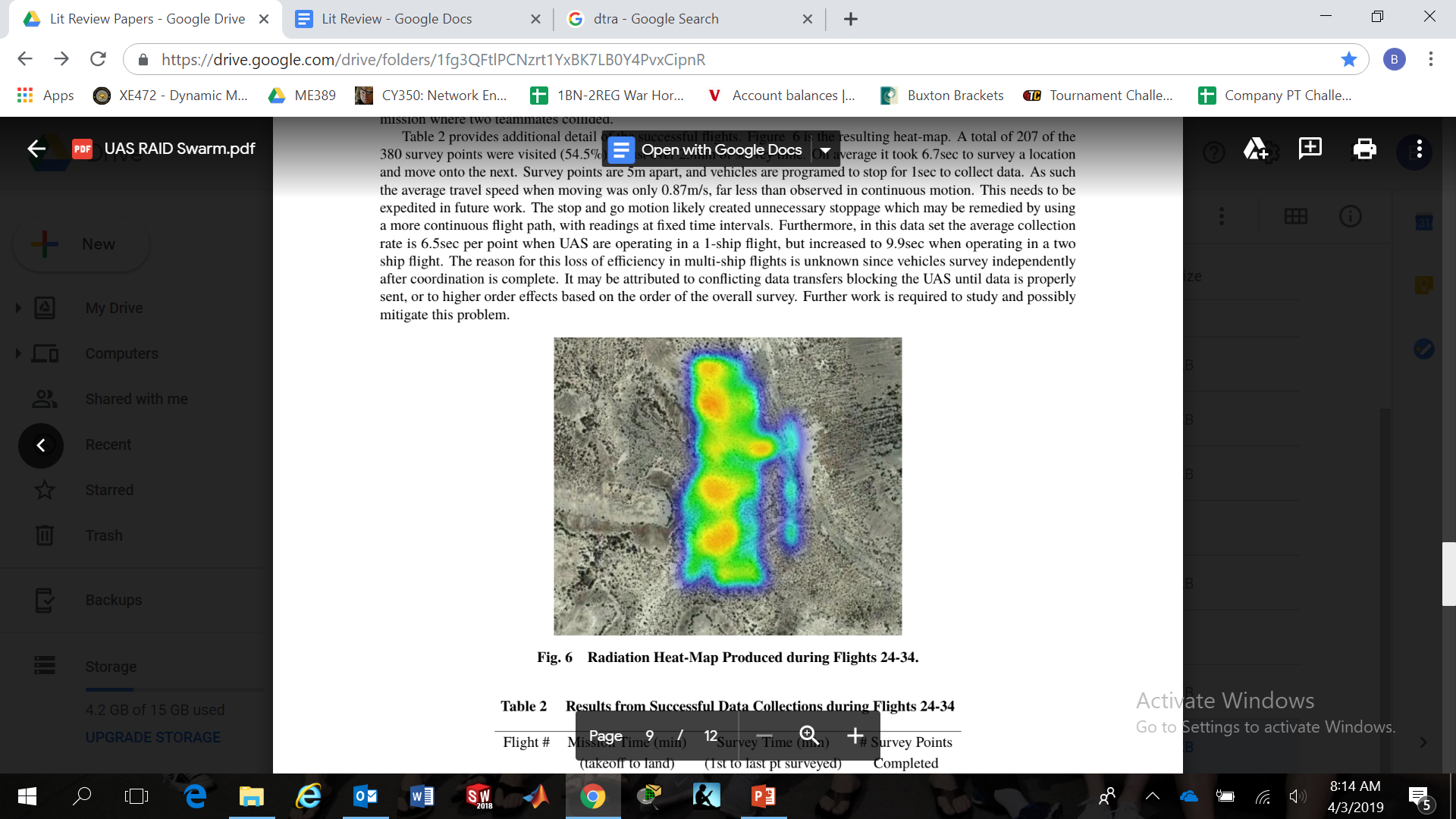


**Figure 5. Running SCRIMMAGE Simulation**

A total of four trials were run, one with 4 UAVs, 3, 2, and finally 1 UAV. The limit was held to 4 UAVs because anything greater significantly impacted the computing power to the point where delays during the re-planning process had time ramifications. This is a current limitation to the algorithm, but due to the rapid technological advances in today’s society, will likely be overcome in the near future (Roser, 2013).

In order to maintain as many constants as possible, the size of the map was held to 25x25 waypoints for all conducted trials. The map was created in a separate file and imported into the Savidge Swarm and Greedy Goto files. This file not only created the map but assigned scores to each point on the map. For the sake of these simulations, one “hotspot” was used for every trial. The hotspot was the waypoint with the highest score, from which scores were propagated outwards, as described in section 3.3. However, note that in this case, the propagated values are not *predicted* values. Instead, they represent actual values being read in by the UAVs in simulation. In creating the heatmap, the hotspot was assigned an arbitrary score of 160, whereas the default score of the map was 10 (see Appendix C). Figure 6 compares an actual heat map to the display of a portion of a simulated heat map using the described methods. The simulated map is inspired by the actual heat map but is not necessarily trying to replicate the same exact readings.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 40 | 80 | 80 | 80 | 40 | 20 | 10 |
| 40 | 80 | 160 | 80 | 40 | 20 | 10 |
| 40 | 80 | 80 | 80 | 40 | 20 | 10 |
| 40 | 40 | 40 | 40 | 40 | 20 | 10 |
| 20 | 20 | 20 | 20 | 20 | 20 | 10 |
| 10 | 10 | 10 | 10 | 10 | 10 | 10 |
| 10 | 10 | 10 | 10 | 10 | 10 | 10 |



(Kopeikin, 2019)

**Figure 6. Comparison of Actual Heat Map (left) to a Section of a Simulated Heat Map (right)**

The data collection process required modifications to both the original Savidge Swarm and Greedy Goto files. It required that data was read to comma-separated list files such that graphs of the read values could be produced, as seen in section 4.2 (raw data) of this paper. Additionally, Greedy Goto needed an additional matrix that stored all of the known values, because it originally only sent its values to a ground station, and then dumped this data from its memory. Appendix A describes the entire list of files involved in running the simulations.

**4.2 Data Analysis**

The figures below summarize the data collected from the four trials conducted for comparison. These trials were conducted in SCRIMMAGE on a 25x25 waypoint heat map, with a maximum reading of 160 and a minimum reading of 10. The variable of interest is the total score recorded by the UAVs. This metric is important because the score is both a combination of the total points read as well as some weighted importance of each point. Since the overall goal is to locate and survey areas of interest as fast as possible, a greater increase in total score represents more points being read in these valued regions.

**Figure 7. Savidge Swarm versus Greedy Goto Comparison for 1 UAV**

**Figure 8. Savidge Swarm versus Greedy Goto Comparison for 2 UAVs**

**Figure 9. Savidge Swarm vs Greedy Goto Comparison for 3 UAVs**

**Figure 10. Savidge Swarm versus Greedy Goto Comparison for 4 UAVs**

Figures 7 through 10 graphically depict the performance of Savidge Swarm algorithm versus Greedy Goto algorithm for trials of 1 UAV to 4 UAVs. In order to quantify the differences, it is important to remember the goal of the new algorithm was to locate areas of interest as fast as possible, not survey the entire map as fast as possible. Thus, instead of looking at how long it takes to obtain the entire score value, the data was analyzed for time to reach 50% of the total score. Percent improvement was calculated using Greedy Goto as the baseline, as according to Equation 3:

, [Equation 3]

where *SS* and *GG* are the times for Savidge Swarm and Greedy Goto to reach 50%, respectively. Table 1 below summarizes these findings for the 4 trials.

**Table 1. Summary of Quantitative Results**

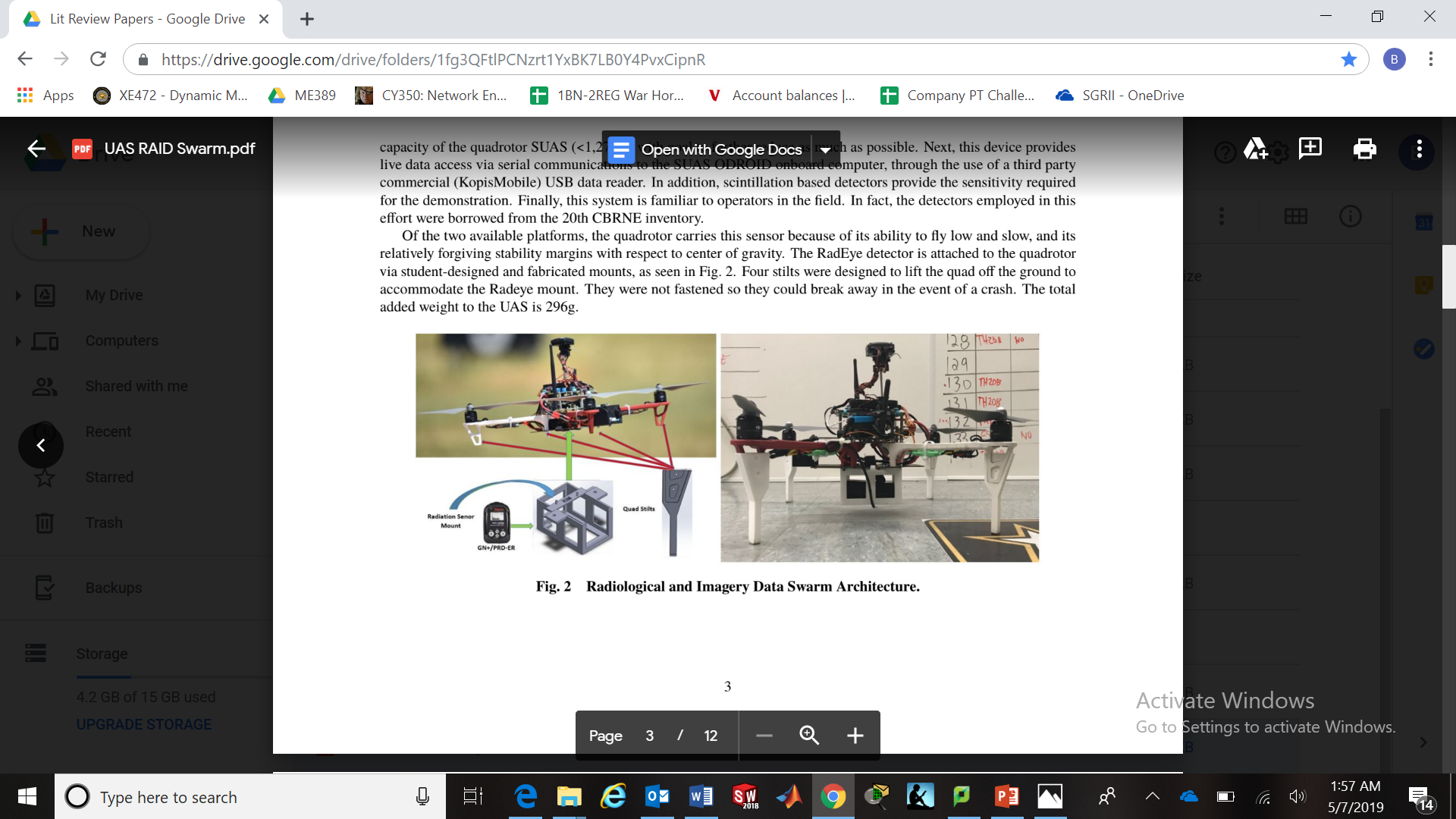
|  |  |  |  |
| --- | --- | --- | --- |
| Number of UAVs | Savidge Swarm to 50% (seconds) | Greedy Goto to 50%  (seconds) | % Improvement |
| 1 | 146.6 | 175.5 | 19.7 |
| 2 | 93.4 | 113.0 | 21.0 |
| 3 | 51.0 | 70.6 | 39.2 |
| 4 | 44.6 | 48.8 | 9.42 |

The data displayed in Table 1 makes it clear that Savidge Swarm does in fact provide improvement over Greedy Goto. Even as the number of UAVs was varied, the algorithm consistently outperformed the baseline using the chosen metric. The trial for 1 UAV was primarily conducted to demonstrate that the re-planning process, involving predicting other values and using cost-benefit analysis, is more effective than the baseline. This is evident in the 19.7% improvement from this trial. The significant decrease in improvement at 4 UAVs can likely be attributed to the fact that in the planning process, UAVs sequentially choose all of their waypoints. Thus, one UAV likely selected all of the points in the area of interest, giving no other UAVs the chance to survey points in that region. This is a problem that is addressed later in section 6.2 (future implications) of this paper.

Looking back to the graphs, it is clear that the total recorded score of Savidge Swarm initially increases more rapidly than Greedy Goto. This is significant because it means the UAVs are getting to the points labelled with higher importance faster than the baseline algorithm is capable of. Although the Greedy Goto total score does eventually rise above the Savidge Swarm, this is not a concern because the goal of the algorithm is not merely to map all of the points. In fact, it is expected Greedy Goto will survey the entire map faster because the UAVs simply move directly from waypoint to waypoint. Meanwhile, Savidge Swarm requires UAVs to gravitate away from the shortest path towards areas of interest, then to return to the less significant waypoints later.

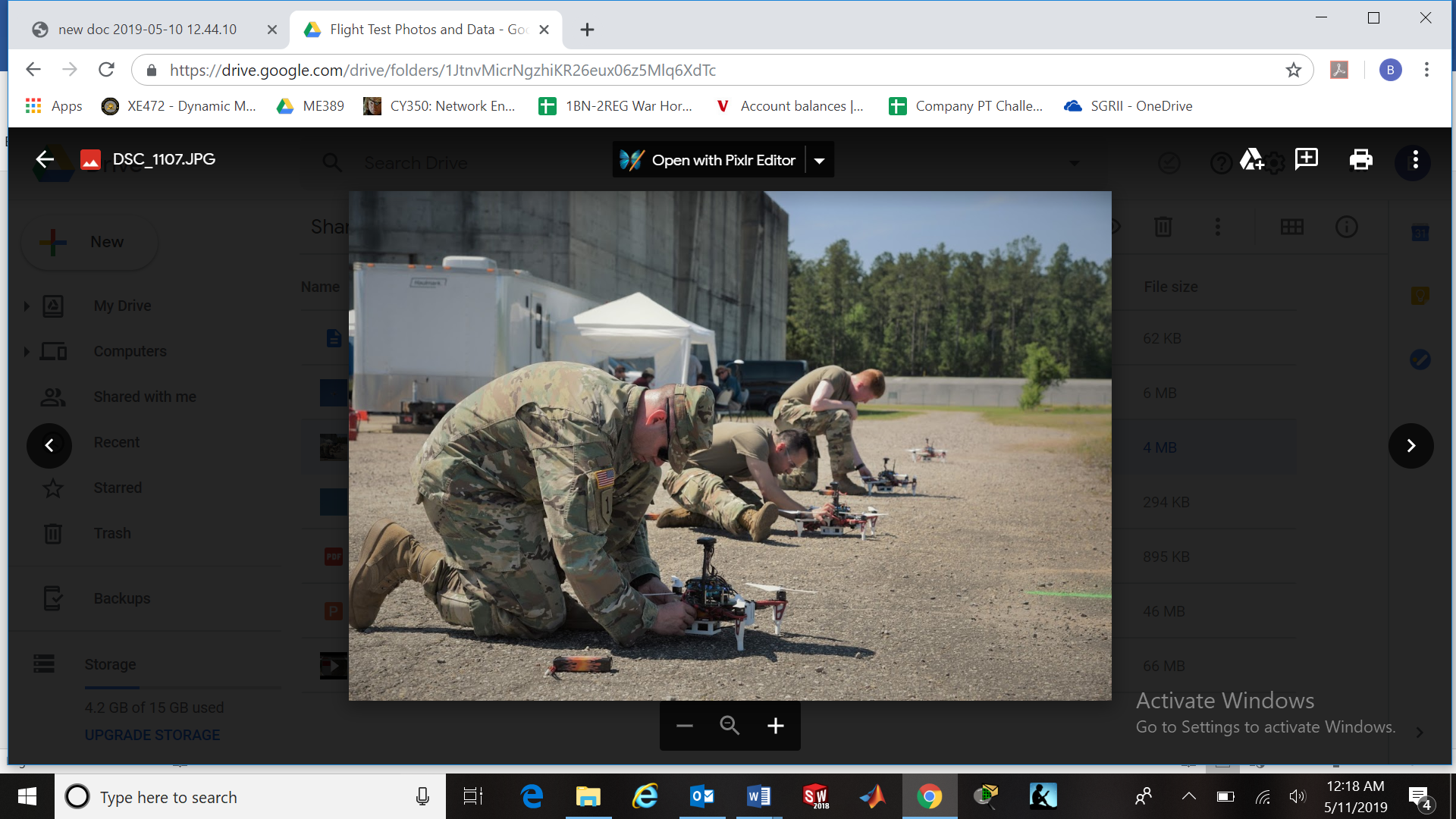
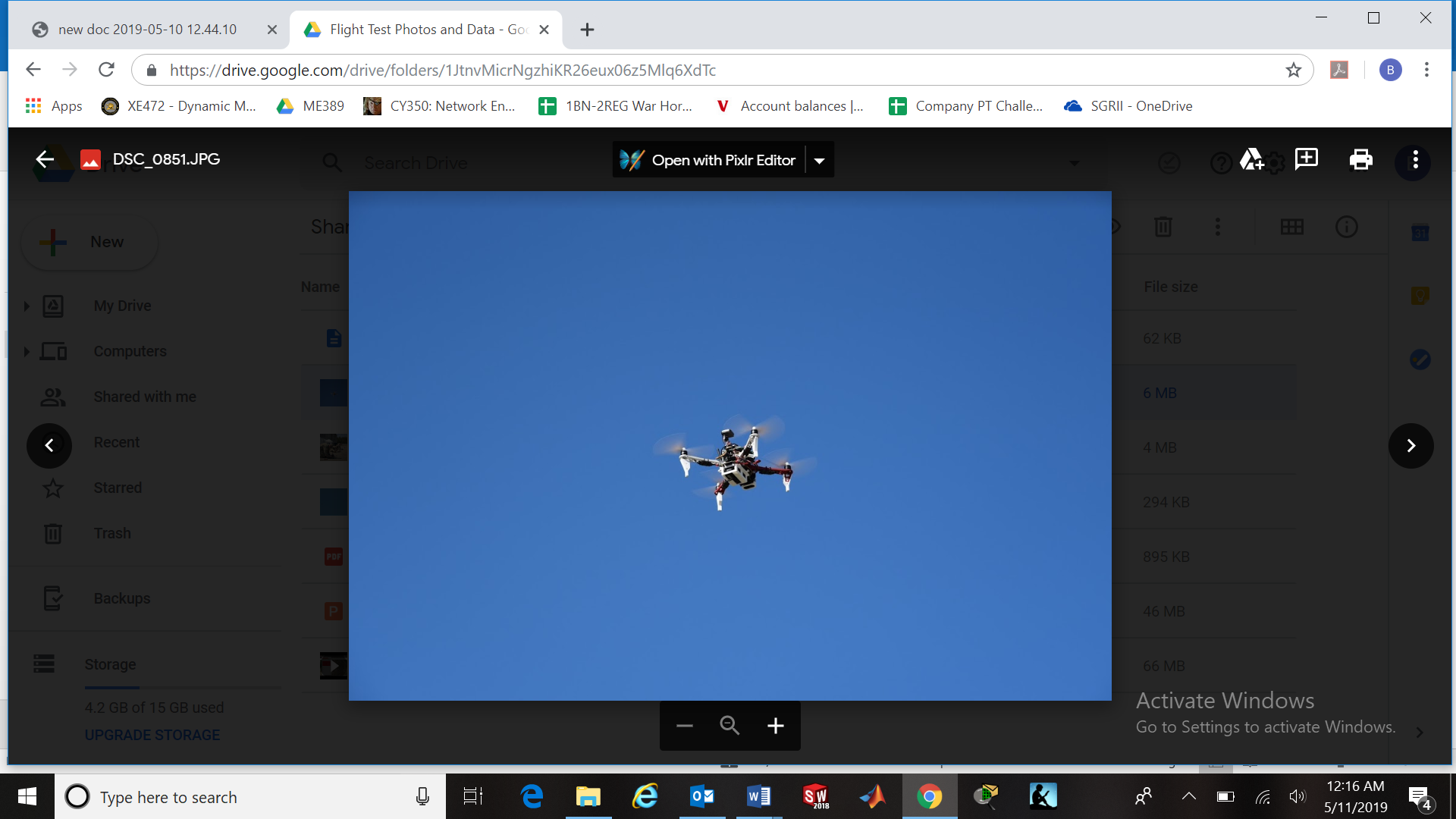
**5. Flight Testing**

Although the primary analysis of the new swarming behavior was conducted in simulation, a physical flight test was still conducted. This required even further modification of the original code to Savidge Swarm in order to implement it onto a physical UAV swarm. It was only a single flight with 2 UAVs in the swarm, but the main takeaway is that the new algorithm can in fact be run on a live swarm. The flight test involved actual radiation sources and actual radiation sensors that recorded and analyzed data, shared it with the other UAV, and then ran through a re-planning process. Figure 11 shows one of the UAVs with a live radiation sensor attached (Kopeikin, 2019).



**Figure 11. UAVs Equipped with Radiation Sensors and Programmed with Swarming Algorithm**

Additionally, the Figure 12 depicts the preparation and live flight testing of the drones.

**Figure 12. Preparation and Live Flight Testing**

6. Conclusion

**6.1 Important Takeaways**

The goal of locating and surveying areas of interest faster than the baseline Greedy Goto algorithm was accomplished in this project. A new algorithm was implemented with UAV-to-UAV communications and periodic re-planning processes in order to redirect towards areas of interest. The new algorithm was tested in simulation using SCRIMMAGE, from which the generated plots revealed Savidge Swarm to outperform the baseline algorithm. Additionally, the new algorithm worked when applied to an actual UAV swarm, providing confidence in future implementation of Savidge Swarm. Although limited computing power sets limits on this algorithm, the rapidly developing technological realm will permit this algorithm to be used on a larger scale in the near future.

**6.2 Future Implications**

Running through multiple simulations revealed many improvements that can be made on the new algorithm but were not implemented due to time considerations. The simplest improvement would be to modify the propagation value. As stated above, the value for these trials was set at 1.3, but it is very possible a different value may be more effective for this algorithm. A smaller value, closer to 1 could benefit in that the propagated values would spread even further. However, too small a value would make all the predicted values too similar. Thus, this parameter is a balance between propagation and differences between values. Another parameter that could easily be altered is the time after which the UAVs re-plan. For these simulations, 10 seconds was used. Given infinite computing power, this value would be smaller, but it can only be set so low before the time taken to compute the re-planning process outweighs the benefit of re-planning. This minimum limit, determined by control chatter, needs to ensure the drones are given enough time to execute the mission, rather than just planning over and over again.

Potentially the biggest flaw with Savidge Swarm is that one UAV at a time plans its entire path before any of the others even get to choose a single point. In the re-planning process, it would be beneficial for each UAV to choose one waypoint at a time, then allowing the other UAVs to choose a waypoint before deciding its next destination. Savidge Swarm limits the ability for other UAVs move into the areas of interest because one UAV adds all of the waypoints in that area to its waypoint list before any of the other UAVs get the chance to move in that direction. Choosing one waypoint at a time would allow more UAVs to converge on the areas of interest.

An additional improvement is to start the UAVs surveying in more scattered locations around the map. Right now, the UAVs are programmed to just pick the nearest point and it generally leads to them all starting on the same side of the map. In Savidge Swarm, the waypoints on the perimeter are already stored in a list. The UAVs could potentially use the length of this list and the number of UAVs to divide in such a way that they start spread out evenly around the perimeter. This would allow the initial search to cover a broader range and more efficiently locate areas of interest.

A final suggestion is to reconsider the cost-benefit analysis. In this algorithm, cost-benefit is set at a one-to-one ratio, according to equation 2. However, it is possible that making the predicted score at a given waypoint more important than the distance required to travel there would make the algorithm more efficient. Implementing this would make use of modified version of Equation 2:

, [Equation 4]

where the addition of the variable *x* serves as a weight for the predicted score relative to the distance. If predicted score is seen as more important, *x* would be greater than 1, whereas distance carrying more weight would require *x* is less than 1. It is important to note this method would only work if the believed values are generally greater than 1, or else trying to weight them higher would actually decrease their importance as a result of the exponent.

**Appendix A. SCRIMMAGE Use**

Steps to run SCRIMMAGE on Linux:

1. cd scrimmage

2. cd scrimmage

3. scrimmage ../usma/missions/(insert .xml file here)

Ex. scrimmage ../usma/missions/savidgeswarm\_simulations.xml

SCRIMMAGE Commands:

‘b’ - Pause/Resume

‘spacebar’ - while paused, increment 0.1 seconds

‘]’ - speed up

‘[‘ - slow down

‘right/left arrow keys’ - switch to other drones’ views

‘a’ - rotate through camera views

List of files for running a simulation:

savidgeswarm\_simulations.xml – file that actually gets run in SCRIMMAGE

-modify: number of UAVs, starting location of UAVs

trial\_enumerations.py – creates heatmap

-modify: grid locations, number of waypoints, distance between waypoints, location of hotspot, score of hotspot, default map score

savidgeswarm\_v2\_simulation.py – contains actual behavior

-modify: time between re-planning, propagation values

Savidgeswarm\_simulations.xml – creates the objects from the class of savidgeswarm\_v2\_simulations

-modify: nothing

swarm\_helper\_v2.py – used for sending messages

-modify: nothing

Simulations.csv – output file

-location: /home/user1/scrimmage/scrimmage

**Appendix B**. **Savidge Swarm Code for SCRIMMAGE**

# Performs a sequencial greedy waypoint (WP) assignment:

# 1- Determine which UAVs are in your subswarm

# 2- Split up number of wpts to be assigned among subswarm

# 3- Each UAS individually plans for all UAS in the subswarm in sequence follows:

# - Determine closest point to that UAS, add that point to your bundle

# - Find next closest point to that point, add to bundle

# - Repeat until bundle is full.

# - As waypoints are added to bundle, make them unavailable to others

# - Repeat this process for each UAS in sequence

# 4- Each UAS then assigns itself its own set of waypoints

# 5- Each UAS is sequenced to navigate through its bundle of waypoints

#

# Brian Savidge

# USMA

# 11 May 2019

import numpy as np

import math

import autopilot\_bridge.msg as apbrg

import ap\_msgs.msg as apmsg

import ap\_lib.swarm\_helper as pursueBytes

from autopilot\_bridge.msg import LLA

import ap\_lib.gps\_utils as gps

import ap\_lib.ap\_enumerations as enums

import ap\_lib.sasc\_scrimmage as ss

import trial\_enumerations as usma\_enums

import timeit

DIST2WP\_QUAD = 10

DIST2WP\_ZEPH = 75

DIST\_START\_DESCENT = 20

TIME\_AT\_WP = 3

SAFE\_ALT = 100

SURVEY\_ALT = 20

class Savidgeswarm(ss.Tactic):

def init(self, params):

self.\_id = int(params['id'])

self.\_target\_id = -1

self.\_wp = np.array([0, 0, 0])

self.\_max\_range = enums.MAX\_RANGE\_DEF

self.\_fov\_width = enums.HFOV\_DEF

self.\_fov\_height = enums.VFOV\_DEF

self.\_own\_pose = apbrg.Geodometry()

self.\_blues = dict()

self.\_shot = set()

self.\_safe\_waypoint = np.array([0, 0, 0])

self.\_last\_ap\_wp = np.array([0, 0, 0])

self.\_action = ["None", 0]

self.\_vehicle\_type = int(params['vehicle\_type'])

self.\_name = 'SavidgeSwarm'

# Initialize Variables for Waypoint Assignment

self.\_subswarm\_id = 0

self.\_id\_in\_subswarm = []

self.\_first\_tick = True

self.\_subswarm\_num\_to\_assign = []

self.\_subswarm\_wp\_assignment = dict()

self.\_wp\_assigned = []

for i in range(0, len(usma\_enums.WP\_LOC)):

self.\_wp\_assigned.append(False)

# Initialize Variables for Sequencing between Waypoints

self.\_wp\_id\_list = [] # List of WPs to be assigned to this UAS

for i in range(0, len(usma\_enums.WP\_LOC)):

self.\_wp\_id\_list.append(i) # Place holder for other logic

self.\_wp\_id\_list\_id = 0 # Index within assigned list of WPs

self.\_loc = self.\_wp\_id\_list[self.\_wp\_id\_list\_id]

self.\_desired\_lat = float(usma\_enums.WP\_LOC[self.\_loc][0])

self.\_desired\_lon = float(usma\_enums.WP\_LOC[self.\_loc][1])

self.\_desired\_alt = self.\_last\_ap\_wp[2]

self.\_original\_alt = SAFE\_ALT

self.\_time\_at\_wp = 0

self.\_time\_start = 0

self.\_at\_wp = False

self.\_belief = [] #Predicted values of waypoints

self.\_previous\_value = 0 #Place holder for future use

self.\_new\_WPs = [] #WPs read in last iteration

self.\_previous\_lowest = 99999

self.\_Known\_WP\_LOC = [] #Known WP values

for i in range(0,len(usma\_enums.WP\_LOC)):

self.\_Known\_WP\_LOC.append(0)

self.\_belief.append(0)

self.\_lat\_points = usma\_enums.lat\_points #Number of latitudinal points

self.\_long\_points = usma\_enums.long\_points #Number of longitudinal points

self.\_perimeter\_WP = [] #Points on the perimeter

for i in range(0,self.\_long\_points):

self.\_perimeter\_WP.append(i)

self.\_perimeter\_WP.append((self.\_lat\_points-1)\*self.\_long\_points+i)

for i in range(1,self.\_lat\_points):

self.\_perimeter\_WP.append(i\*self.\_long\_points)

self.\_perimeter\_WP.append(i\*self.\_long\_points+self.\_long\_points-1)

self.\_time = 0 #Keeps track of time starting from 0

self.\_num\_UAVs = 0 #Initialized for later use

self.\_blue\_in\_subswarm = dict() #Initialized for later use

self.\_propogation\_decrease = 1.3 #Decrease from one WP to the next

self.pursuit\_status = True #For later use

#For sending and receiving messages

try:

topic = 'network/send\_swarm\_behavior\_data'

# we dont need to create a subscriber here because this is taken care of in

self.pubs[topic] = self.\_parent.\_behavior\_data\_publisher

except AttributeError:

self.pubs[topic] = \

self.createPublisher(topic, apmsg.BehaviorParameters, 1)

self.subs[topic] = \

self.createSubscriber(

topic, apmsg.BehaviorParameters, self.\_process\_swarm\_data\_msg

)

#Function for sending messages

def pursuit\_status\_update(self):

parser = pursueBytes.PursuitMessageParser()

parser.friendly\_id = self.\_id

parser.target\_id = self.\_loc

parser.target\_distance = self.\_Known\_WP\_LOC[self.\_loc]

parser.pursuit\_status = True

data = parser.pack()

net\_msg = apmsg.BehaviorParameters()

net\_msg.id = pursueBytes.PURSUIT\_MESSAGE

net\_msg.params = data

#for \_ in range(1): # Hope to get at least one through

self.pubs['network/send\_swarm\_behavior\_data'].publish(net\_msg)

def step\_autonomy(self, t, dt):

# Execute this portion of the code on the loop only

if self.\_first\_tick == True:

self.\_time = timeit.default\_timer() #Sets start time

self.\_first\_tick = False #Makes sure this only happens once

# Set initial altitude settings

self.\_desired\_alt = self.\_last\_ap\_wp[2]

self.\_original\_alt = self.\_last\_ap\_wp[2]

# Determine your own subswarm ID

#key #value

for blue\_id, blue in self.\_blues.iteritems():

if blue.vehicle\_id == self.\_id:

self.\_subswarm\_id = blue.subswarm\_id

break

print "subswarm\_id: ", self.\_subswarm\_id

# Build a list of all vehicle IDs within your subswarm

blue\_in\_subswarm = dict()

i = 0

for blue\_id, blue in self.\_blues.iteritems():

if blue.subswarm\_id == self.\_subswarm\_id:

self.\_id\_in\_subswarm.append(blue.vehicle\_id)

self.\_subswarm\_num\_to\_assign.append(0)

blue\_in\_subswarm[i] = blue

self.\_blue\_in\_subswarm[i] = blue

i = i+1

print "id\_in\_subswarm: ", self.\_id\_in\_subswarm

# Divide # of waypoints by # of vehicles and create empty bundle of wpts for each

num\_in\_subwarm = len(self.\_id\_in\_subswarm)

self.\_num\_UAVs = num\_in\_subwarm

for i in range(0, num\_in\_subwarm):

self.\_subswarm\_num\_to\_assign[i] = int(math.floor(len(usma\_enums.WP\_LOC)/num\_in\_subwarm))

if i < (len(usma\_enums.WP\_LOC) % num\_in\_subwarm):

self.\_subswarm\_num\_to\_assign[i] = self.\_subswarm\_num\_to\_assign[i] + 1

print "num\_to\_assign: ", self.\_subswarm\_num\_to\_assign

# Perform sequencial greedy wpt assignment. Loop over each UAS in subswarm.

for i in range(0, num\_in\_subwarm):

# Set the start location to current UAS position

temp\_lat = blue\_in\_subswarm[i].state.pose.pose.position.lat

temp\_lon = blue\_in\_subswarm[i].state.pose.pose.position.lon

assignment\_list = []

# Loop over each element of the waypoint bundle

for j in range(0, self.\_subswarm\_num\_to\_assign[i]):

min\_dist = 99999 #initialize as large number

new\_wp\_assigned = False

# Loop over each waypoint defined in the mission

for k in range(0, len(usma\_enums.WP\_LOC)):

# Skip to next if that waypoint is already assigned

if self.\_wp\_assigned[k] == False:

# Set the end location to that waypoint

temp2\_lat = float(usma\_enums.WP\_LOC[k][0])

temp2\_lon = float(usma\_enums.WP\_LOC[k][1])

# Check if start to end location distance is new minimum, if so mark

# it for assignment

temp\_dist = gps.gps\_distance(temp\_lat, temp\_lon, temp2\_lat, temp2\_lon)

if temp\_dist < min\_dist:

min\_dist = temp\_dist

wp\_to\_assign = k

new\_wp\_assigned = True

# Add the next closest waypoint to the bundle

if new\_wp\_assigned == True:

assignment\_list.append(wp\_to\_assign)

self.\_subswarm\_wp\_assignment[i] = assignment\_list

# Mark that waypoint as "assigned" so unavailable to others

self.\_wp\_assigned[wp\_to\_assign] = True

# Update the start location to that waypoint

temp\_lat = float(usma\_enums.WP\_LOC[wp\_to\_assign][0])

temp\_lon = float(usma\_enums.WP\_LOC[wp\_to\_assign][1])

# Assign yourself your own bundle of waypoints

if blue\_in\_subswarm[i].vehicle\_id == self.\_id:

self.\_wp\_id\_list = self.\_subswarm\_wp\_assignment[i]

print "subswarm\_wp\_assignment: ", self.\_subswarm\_wp\_assignment

# Proceed to the first Waypoint in the bundle

self.\_loc = self.\_wp\_id\_list[0]

self.\_desired\_lat = float(usma\_enums.WP\_LOC[self.\_loc][0])

self.\_desired\_lon = float(usma\_enums.WP\_LOC[self.\_loc][1])

print "Going to wp: ", self.\_loc

# Go to desired latitude, longitude, and maintain altitude

# deconfliction:

self.\_wp = np.array([self.\_desired\_lat, self.\_desired\_lon,

self.\_desired\_alt])

pos = self.\_own\_pose.pose.pose.position

dist = gps.gps\_distance(pos.lat, pos.lon, self.\_desired\_lat, self.\_desired\_lon)

# Detect whether UAS has arrived at WP (within threshold distance), track time at WP

# Zephyrs (type 2) loiter around point, so set threshold distance > loiter radius

# Set threshold distance for Quads (type 1), much smaller

if (self.\_vehicle\_type == 2 and dist < DIST2WP\_ZEPH) or (self.\_vehicle\_type == 1 and dist < DIST2WP\_QUAD):

if self.\_at\_wp == False:

self.\_time\_start = timeit.default\_timer()

self.\_at\_wp = True

self.\_time\_at\_wp = timeit.default\_timer() - self.\_time\_start

else:

self.\_at\_wp = False

self.\_time\_at\_wp = 0

# Detect if Quad is close enough to first WP to descend to survey altitude

if self.\_vehicle\_type == 1 and dist < DIST\_START\_DESCENT:

self.\_desired\_alt = SURVEY\_ALT

# After X time has elapsed at WP, move onto next WP in your bundle

if self.\_time\_at\_wp > TIME\_AT\_WP:

self.\_Known\_WP\_LOC[self.\_loc] = usma\_enums.WP\_LOC[self.\_loc][2] #Reads value at WP

self.\_new\_WPs.append(self.\_loc) #Adds WP to newly read list

self.pursuit\_status\_update() #Sends message to update other UAVs

self.\_loc = self.\_wp\_id\_list[self.\_wp\_id\_list\_id]

self.\_wp\_id\_list\_id = self.\_wp\_id\_list\_id + 1

# If you get to the end of your bundle, repeat from its beginning

# and reset to original altitude

if self.\_wp\_id\_list\_id > (len(self.\_wp\_id\_list)-1):

self.\_wp\_id\_list\_id = 0

self.\_desired\_alt = self.\_original\_alt

self.\_loc = self.\_wp\_id\_list[self.\_wp\_id\_list\_id]

print "Loc " + repr(self.\_loc)

self.\_desired\_lat = float(usma\_enums.WP\_LOC[self.\_loc][0])

self.\_desired\_flon = float(usma\_enums.WP\_LOC[self.\_loc][1])

# Reset these so that UAV knows it's no longer at its goal WP

self.\_at\_wp = False

self.\_time\_at\_wp = 0

print "Going to wp: ", self.\_loc

if timeit.default\_timer()-self.\_time >= 10: #The process will update every number of specified seconds

#First need to update beliefs based on newly read values

for new\_wp in range(0,len(self.\_new\_WPs)):

q = 1 #Keeps track of how many WPs belief's update

i = 1 #number of WPs away from center

while q > 0:

q = 0

for j in range(-i,i+1):

for k in range(-i,i+1):

outside = "False"

#These checks ensure we're not propogating outside of the perimeter

if j <= 0:

if k <= 0:

for outsidek in range(k,1):

for outsidej in range(j,1):

if new\_wp+outsidek\*self.\_long\_points+outsidej in self.\_perimeter\_WP or new\_wp+outsidek\*self.\_long\_points+outsidej < 0 or new\_wp+outsidek\*self.\_long\_points+outsidej > len(self.\_Known\_WP\_LOC)-1:

outside = "True"

elif k > 0:

for outsidek in range(1,k+1):

for outsidej in range(j,1):

if new\_wp+outsidek\*self.\_long\_points+outsidej in self.\_perimeter\_WP or new\_wp+outsidek\*self.\_long\_points+outsidej < 0 or new\_wp+outsidek\*self.\_long\_points+outsidej >= len(self.\_Known\_WP\_LOC)-1:

outside = "True"

elif j > 0:

if k <= 0:

for outsidek in range(k,1):

for outsidej in range(1,j+1):

if new\_wp+outsidek\*self.\_long\_points+outsidej in self.\_perimeter\_WP or new\_wp+outsidek\*self.\_long\_points+outsidej < 0 or new\_wp+outsidek\*self.\_long\_points+outsidej > len(self.\_Known\_WP\_LOC)-1:

outside = "True"

elif k > 0:

for outsidek in range(1,k+1):

for outsidej in range(1,j+1):

if new\_wp+outsidek\*self.\_long\_points+outsidej in self.\_perimeter\_WP or new\_wp+outsidek\*self.\_long\_points+outsidej < 0 or new\_wp+outsidek\*self.\_long\_points+outsidej > len(self.\_Known\_WP\_LOC)-1:

outside = "True"

#Update propogated belief as long as it's not outside perimeter and greater than previous belief

if outside == "False":

if self.\_Known\_WP\_LOC[new\_wp]/self.\_propogation\_decrease\*\*i > self.\_belief[new\_wp+self.\_long\_points\*k+j]:

self.\_belief[new\_wp+self.\_long\_points\*k+j] = self.\_Known\_WP\_LOC[new\_wp]/self.\_propogation\_decrease\*\*i

q = q+1

i = i+1

self.\_new\_WPs = [] #Resets the newly read WPs array so they don't later get double counted

#For setting loewst read value to the rest of the WP belief values

lowest\_WP\_value = 99999

for i in range(0,len(usma\_enums.WP\_LOC)):

if self.\_Known\_WP\_LOC[i] < lowest\_WP\_value:

lowest\_WP\_value = self.\_Known\_WP\_LOC[i]

for i in range(0,len(usma\_enums.WP\_LOC)):

if self.\_belief[i] == self.\_previous\_lowest:

self.\_belief[i] = lowest\_WP\_value

self.\_previous\_lowest = lowest\_WP\_value

#Then update behavior based on new beliefs

# Perform sequencial greedy wpt assignment. Loop over each UAS in subswarm.

for i in range(0,self.\_num\_UAVs):

self.\_subswarm\_num\_to\_assign[i] = 0

self.\_subswarm\_wp\_assignment = dict()

self.\_wp\_id\_list = [] # List of WPs to be assigned to this UAS

for i in range(0, len(usma\_enums.WP\_LOC)):

self.\_wp\_id\_list.append(i)

self.\_wp\_assigned[i] = False

assign\_WPs = 0

#Check which WPs haven't been read:

for x in range(0,len(self.\_Known\_WP\_LOC)):

if self.\_Known\_WP\_LOC[x] == 0:

assign\_WPs = assign\_WPs+1

#These 2 lines intended to prevent issue of running out of WPs

if assign\_WPs < self.\_num\_UAVs:

return True

for i in range(0, self.\_num\_UAVs):

self.\_subswarm\_num\_to\_assign[i] = int(math.floor(assign\_WPs/self.\_num\_UAVs))

if i < (assign\_WPs % self.\_num\_UAVs):

self.\_subswarm\_num\_to\_assign[i] = self.\_subswarm\_num\_to\_assign[i] + 1

print self.\_subswarm\_num\_to\_assign[i]

for i in range(0, self.\_num\_UAVs):

# Set the start location to current UAS position

temp\_lat = self.\_blue\_in\_subswarm[i].state.pose.pose.position.lat

temp\_lon = self.\_blue\_in\_subswarm[i].state.pose.pose.position.lon

assignment\_list = []

# Loop over each element of the waypoint bundle

for j in range(0, self.\_subswarm\_num\_to\_assign[i]):

max\_score = -1 #initialize as -1

new\_wp\_assigned = False

# Loop over each waypoint defined in the mission

for k in range(0, len(usma\_enums.WP\_LOC)):

# Skip to next if that waypoint is already assigned

if self.\_wp\_assigned[k] == False and self.\_Known\_WP\_LOC[k] == 0:

# Set the end location to that waypoint

temp2\_lat = float(usma\_enums.WP\_LOC[k][0])

temp2\_lon = float(usma\_enums.WP\_LOC[k][1])

# Check if score of WP is better than max, if so mark it for assignment

temp\_dist = gps.gps\_distance(temp\_lat, temp\_lon, temp2\_lat, temp2\_lon)

if temp\_dist == 0:

max\_score = 999999999

wp\_to\_assign = k

new\_wp\_assigned = True

elif self.\_belief[k]/temp\_dist > max\_score:

max\_score = self.\_belief[k]/temp\_dist

wp\_to\_assign = k

new\_wp\_assigned = True

# Add the next highest score waypoint to the bundleb

if new\_wp\_assigned == True:

assignment\_list.append(wp\_to\_assign)

self.\_subswarm\_wp\_assignment[i] = assignment\_list

# Mark that waypoint as "assigned" so unavailable to others

self.\_wp\_assigned[wp\_to\_assign] = True

# Update the start location to that waypoint

temp\_lat = float(usma\_enums.WP\_LOC[wp\_to\_assign][0])

temp\_lon = float(usma\_enums.WP\_LOC[wp\_to\_assign][1])

# Assign yourself your own bundle of waypoints

if self.\_blue\_in\_subswarm[i].vehicle\_id == self.\_id:

self.\_wp\_id\_list = self.\_subswarm\_wp\_assignment[i]

print "subswarm\_wp\_assignment: ", self.\_subswarm\_wp\_assignment

# Proceed to the first Waypoint in the bundle

self.\_loc = self.\_wp\_id\_list[0]

self.\_wp\_id\_list\_id = 0

self.\_desired\_lat = float(usma\_enums.WP\_LOC[self.\_loc][0])

self.\_desired\_lon = float(usma\_enums.WP\_LOC[self.\_loc][1])

print "Going to wp: ", self.\_loc

#Update self.\_time parameter to reset to determine next time interval

self.\_time = timeit.default\_timer()

print "This is the file"

return True

#To receive and process messages

def \_process\_swarm\_data\_msg(self, msg):

print "Message Recieved!"

if msg.id == pursueBytes.PURSUIT\_MESSAGE:

parser = pursueBytes.PursuitMessageParser()

parser.unpack(msg.params)

newly\_read\_wp = parser.target\_id

self.\_Known\_WP\_LOC[newly\_read\_wp] = parser.target\_distance

self.\_new\_WPs.append(newly\_read\_wp)

**Appendix C. Code for Creating Heatmap for SCRIMMAGE Simulations**

#!/usr/bin/env python

import math

WP\_LOC = []

#How far apart desired waypoints are in meters

mesh = 5

#For creating desired number of waypoints

wp = 25

min\_latitude = 41

#Degrees latitde per meter (do not alter)

lat\_dis = 0.000009005401

#Using 19 for the purpose of this trial, actual max value can be used for real test

max\_latitude = min\_latitude+wp\*mesh\*lat\_dis

if abs((max\_latitude-min\_latitude)/(lat\_dis\*mesh)) % 1 < 0.5:

lat\_points = abs(int((max\_latitude-min\_latitude)/(lat\_dis\*mesh)))

else:

lat\_points = 1+abs(int((max\_latitude-min\_latitude)/(lat\_dis\*mesh)))

print("Latitude points: " + repr(lat\_points))

min\_longitude = -74

#Degrees longitude per meter (do not alter)

long\_dis = 1/(math.cos(min\_latitude\*180/math.pi)\*69.172\*1609.34)

max\_longitude = -74+wp\*mesh\*long\_dis

if abs((max\_longitude-min\_longitude)/(long\_dis\*mesh)) % 1 < 0.5:

long\_points = abs(int((max\_longitude-min\_longitude)/(long\_dis\*mesh)))

else:

long\_points = 1+abs(int((max\_longitude-min\_longitude)/(long\_dis\*mesh)))

print("Longitude points: " + repr(long\_points))

#Default value set for waypoins

default\_value = 10

for i in range(0,lat\_points):

for j in range(0,long\_points):

n = i\*long\_points+j

waypoint = []

waypoint.append(min\_latitude+lat\_dis\*i\*mesh)

waypoint.append(min\_longitude+long\_dis\*j\*mesh)

waypoint.append(default\_value)

waypoint.append(n)

WP\_LOC.append(waypoint)

#print("Waypoint: " + repr(WP\_LOC[n]))

print("Total number of waypoints generated: " + repr(len(WP\_LOC)))

hotspot\_value = 160

hotspot\_lat = 6

hotspot\_long = 10

hotspot\_wp = hotspot\_lat\*long\_points+hotspot\_long

n = 1

decrease = 2

WP\_LOC[hotspot\_wp][2] = hotspot\_value

while hotspot\_value/decrease\*\*n > default\_value:

for i in range(-n,n+1):

for j in range(-n,n+1):

wp\_update = int(hotspot\_wp+long\_points\*i+j)

#print(wp\_update)

if WP\_LOC[wp\_update][2] < int(hotspot\_value/decrease\*\*n):

WP\_LOC[wp\_update][2] = int(hotspot\_value/decrease\*\*n)

n = n+1

for i in range(0,len(WP\_LOC)-1):

print(WP\_LOC[i][2])

# 6. References

Balch, T. and Alkin, R. “Behavior-Based Formation Control for Multirobot Teams.” *IEEE Transcriptions on Robotics and Automation*, vol. 14 (1998). Web. Accessed 18 January 2019.

Burgard, W., Moors, M., Dieter, F., Simmons, R., and Thrun, S. “Collaborative Mutli-Robot Exploration.” *IEEE Transcriptions on Robotics and Automation* (2000). Web. Accessed 11 February 2019.

Cook, Z. “Radiative Contour Mapping Using UAS Swarm.” *UNLV Theses, Dissertations, Professional Papers, and Capstones* (2017). Web. Accessed 18 January 2019.

Eberhart, R., and Shi, Y. “Particle Swarm Optimization: Developments, Applications and Resources.” *IEEE* (2001). Web. Accessed 18 January 2019.

Kopeikin, A., Heider, S., Larkin, D., Bluman, J., Korpela, C., and Morales, R. “Unmanned Aircraft System Swarm for Radiological and Imagery Data Collection.” *United States Military Academy*. West Point, NY. Web. Accessed 18 January 2019.

Kristensen, Hans, and Korda, M. “Status of World Nuclear Forces.” *Federation Of American Scientists*. April 2019. Web. Accessed 07 May 2019.

McCune, R., and Madey, G. “Swarm Control of UAVs for Cooperative Hunting with DDDAS.” *IEEE* (2013)..Web. Accessed 29 January 2019.

Roser, M., and Ritchie, H. “Technological Progress.” *Our World in Data*. 11 May 2013. Web. Accessed 07 May 2019.

“Scrimmage - multi-agent robot simulator.” *GTRI*. [Online]. Available: https://www.scrimmagesim.org/.

Sheng, W., Quingyan, Y., Jindong, T., and Ning, X. “Distributed multi-robot coordination in area exploration.” *Robotics and Autonomous Systems*, vol. 54 (2006). Web. Accessed 11 February 2019.

Simmons, R., Apfelbaum, D., Burgard, W., Fox, D., Moors, M., Thrun, S., and Younes, H. “Coordination for Multi-Robot Exploration and Mapping.” *IEEE Transcriptions on Robotics and Automation* (2000). Web. Accessed 11 February 2019.