3-23-2018

Design and Test of a UAV Swarm Architecture over a Mesh Ad-Hoc Network

Timothy J. Allen

DESIGN AND TEST OF A UAV SWARM ARCHITECTURE OVER A MESH AD-HOC NETWORK

THESIS

Timothy J. Allen, Captain, USAF

AFIT-ENV-MS-18-M-172

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT-ENV-MS-18-M-172

DESIGN AND TEST OF A UAV SWARM ARCHITECTURE OVER A MESH AD-HOC NETWORK

THESIS

Presented to the Faculty

Department of Systems Engineering and Management

Graduate School of Engineering and Management

Air Force Institute of Technology

Air University

Air Education and Training Command

In Partial Fulfillment of the Requirements for the

Degree of Master of Science in Systems Engineering

Timothy J. Allen, BS

Captain, USAF

March 2015

AFIT-ENV-MS-18-M-172

DESIGN AND TEST OF A UAV SWARM ARCHITECTURE OVER A MESH AD-HOC NETWORK

Timothy J. Allen, BS

Captain, USAF

Committee Membership:

Dr. John Colombi
Chair

Dr. David Jacques
Member

Lt Col Amy Cox, PhD
Member

AFIT-ENV-MS-18-M-172

Abstract

Unmanned aerial vehicles (UAV) are individually versatile machines due to their small size, well-developed autopilots, on-board processing, and accurate navigation systems. They can fulfill various military objectives without risking additional manpower including: real-time intelligence (ISR), battle damage assessment (BDA), force application, and force protection.

The purpose of this research was to develop a testable swarm architecture such that the swarm of UAVs collaborate as a team rather than acting as several independent vehicles. Commercial-off-the-shelf (COTS) components were used as they were low-cost, readily available, and previously proven to work with at least two networked UAVs.

Initial testing was successfully performed via software-in-the-loop (SITL) demonstrating swarming of three simulated multirotor aircraft, then transitioned to real hardware. The architecture was then tested in a nylon netting enclosure. Command and control (C2) was provided by software implementing an enhanced version of Reynolds' flocking rules via an onboard companion computer, and UDP multicast messages over a Wi-Fi mesh ad-hoc network. Experimental results indicate a standard deviation between vehicles of 2 meters or less, at airspeeds up to 2 meters per second. This aligns with navigation instrumentation error, permitting safe operation of multiple vehicles within 5 meters of each other. Qualitative observations indicate this architecture is robust enough to handle more aircraft, pass sensor data, and incorporate different swarming algorithms and missions.

Acknowledgments

I would like to express my sincere appreciation to my faculty advisor, Dr. John Colombi, for his guidance and support throughout the course of this thesis effort. His insight and experience were invaluable. More thanks go to the ANT UAV Lab technicians who were critical to supporting flying operations; I'd never have been able to complete real-world testing without them. For keeping me focused during long coding sessions, I'd like to thank the owner of the "Justicepianosongs" channel on YouTube. I would also like to thank my wife and children for their support and encouragement.

Timothy J. Allen

Table of Contents

List of Figures

Page

List of Tables

Page

DESIGN AND TEST OF A UAV SWARM ARCHITECTURE OVER A MESH AD-HOC NETWORK

I. Introduction

General Issue

The history of unmanned aerial vehicles (UAV) is older than manned aircraft; the first hot-air and lighter-than-air balloons were tested without human occupants in the 1700's (Crouch, 2009; PAF, n.d.). The first recorded use of untethered UAVs in a military application is recorded in the March 1849 issue of Scientific American, when the Austrians used balloons to drop bombs on Venice, which had revolted and had no nearby terrain suitable for conventional bombardment (McDaid; Oliver; Strong; Israel, 2002). Interestingly this is also the first recorded instance of any aerial bombardment. Other instances of unmanned aircraft occurred as technology progressed (Fahrney, 1980), but it wasn't until the advent of the microprocessor that UAVs truly became capable of fulfilling missions traditionally performed by manned aircraft (Newcome, 2004).

The modern individual UAV can perform several roles including real time surveillance, battle damage assessment (BDA), target lasing, accurate missile or rocket launch (USAF, 2015a, 2015b), and/or ferry supplies (Lockheed-Martin, 2017). UAVs perform some tasks autonomously, and others as commanded by a man-in-the-loop (Howard, 2013). Coordination with other aircraft, manned and unmanned, is still executed by humans (DOD, 2014). Like humans, unmanned vehicles can synergistically increase mission performance when acting in teams rather than individuals or groups of individuals (Hambling, 2016). Collaborative communication between UAVs is the next hurdle for

UAV technology to increase performance; however, due to ongoing development of swarming networks and how humans can interact with and control them, that hurdle has yet to be crossed (Cummings, Bruni, Mercier, & Mitchell, 2007).

The principles of autonomy exist in a spectrum of flexibility in three categories: task, cognition, and peer/subordinate/supervisor (Rogers, 2011; Scharre, 2015). The work encompassed by this thesis primarily falls under peer flexibility. Peer flexibility encompasses supervisor, subordinate, and peer relationships. To clarify the definition of these roles: autonomous agents filling any of these roles share information with, and receive information from, other agents. Subordinates receive direction or commands from other supervisors, supervisors send direction or commands to subordinates, and peers operate within the confines of their programming given the data shared by other agents. These clear delineations are a starting point but may be insufficient to fully describe swarm behavior as will be shown later.

Problem Statement

There does not exist a flexible architecture to flight test UAV swarms that allows for supervisor/subordinate role reversals. Almost all operational systems place the autonomous agent in the role of subordinate – it exists to execute human decisions.

A notable exception is the UK-made Brimstone missile which can search for targets, select one, and attempt to destroy it with no human input once fired (Marsh, 2014). Lethal systems that can make decisions that end lives without a human operator in the loop are controversial on many levels (Marsh, 2014; Rogers, 2011), but there are more benign tasks like ISR (Saska et al., 2016), refueling (Burns, 2007), and tight formation flying (Justh, EW; Krishnaprasad, 2002), that could be conducted by autonomous decision-making systems.

Some role-reversing autonomous agents have gone operational but by and large they are still in development. The most notable operational example is the F-16's automatic ground collision avoidance system (GCAS). When the flight computer determines the aircraft's trajectory is going to end in a ground collision, at some threshold it takes control from the pilot and performs an emergency recovery maneuver to prevent collision (Norris, 2016). The pilot temporarily becomes the subordinate and the flight computer the supervisor. The F-16 is not normally flown as a UAV, but the supervisor/subordinate roles apply the same way. Modern UAV autopilots fly waypoint or loiter routes as directed by humans in the loop. Algorithms can be used to choose those waypoints and the on-board computer does most of the work of flying but the decision to execute is still the human's: the human is always the supervisor and the UAV the subordinate. One of the drivers for this thesis was to provide a flexible architecture with which to test unmanned systems that allow for similar role changes in a safe and controlled manner.

Research Objectives

The purpose of this research is the enabling of a swarm of three or more multirotor UAVs to act together using collaborative coordination amongst all UAVs in the swarm, without commands from a ground station, except for manual commands to a single "lead" vehicle. A second objective was to investigate the vehicle spacing distribution to determine how closely UAVs can operate in proximity to other UAVs in the swarm while minimizing the chance of colliding, given various flight patterns. For safety purposes and due to institutional requirements, a ground control station (GCS) was used to monitor UAVs on a one-for-one basis, along with an observer and a safety pilot with the ability to immediately seize manual control for each vehicle.

The data produced by this research, and future data acquired by more rigorous testing of the architecture should provide evidence that a three-to-one crew is not required for all UAV tests and operations if the vehicles meet some level of autonomous swarming capability. Metrics that demonstrate safety of flight for a UAV swarm have not been identified at this point in time, and this research should provide some options or insights to develop those metrics.

Investigative Questions

This thesis research focuses on integrating COTS hardware and software prototypes into a collaborative multirotor UAV swarm. The following questions are examined:

- What is one architecture that supports collaborative communication between three or more multirotor air vehicles that can be scaled to include more?
- What is the distribution of separation distances and error between vehicles implementing a version of Reynolds' flocking rules and how does it change with different parameter settings and flight patterns?
- What is the contribution of velocity commands by rule using a "prioritized velocity bucket" instead of a more traditional weighted-rule method?

Methodology

Utilizing COTS equipment and Open Source Software (OSS), can a single operator safely and successfully provide command and control for a swarm consisting of three or more UAVs? The UAVs must form an ad hoc network, where each aircraft is a node and can enter and leave the network freely. Furthermore, the UAVs must act in concert, using onboard processors and telemetry broadcast by each aircraft over the network with no inputs from ground-based command and control (C2). The specific task carried out by the swarm is not of interest in this research, but rather the underlying communication and navigation architecture that enables it. An algorithm mimicking a flock of birds (Reynolds, 1987) will be utilized as a stand-in, applying three control rules to each swarm member plus a fourth rule establishing a desired (safe) minimum altitude.

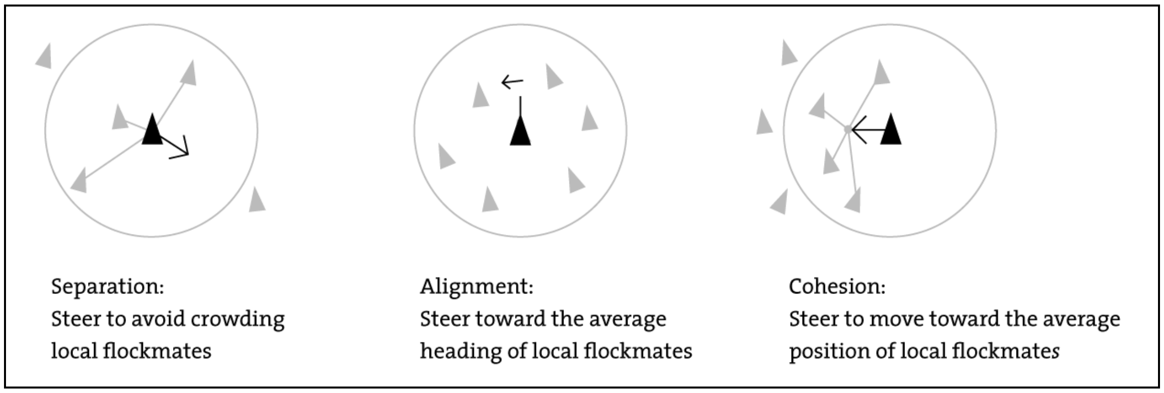

Separation:
Steer to avoid crowding local flockmates
Alignment:
Steer toward the average heading of local flockmates
Cohesion:
Steer to move toward the average position of local flockmates

Figure 1. Reynolds Flocking Rules (Enrica, 2016)

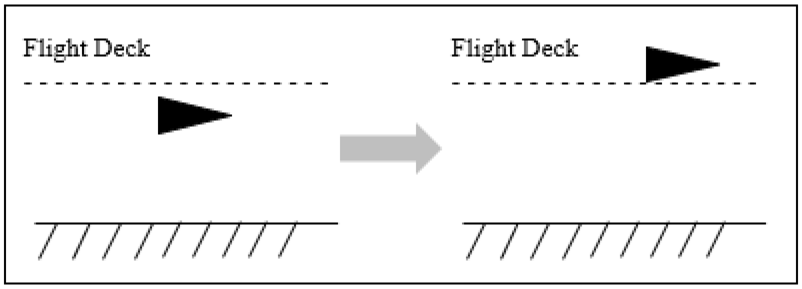


Figure 2. Minimum Altitude Rule

Reynolds' rules by themselves are insufficient to demonstrate peer flexibility because by the previously-made role definitions, all the agents are peers and there are no supervisors or subordinates. Therefore, any given agent must also be able to take on the role of a supervisor where it continues to send and receive information as part of the swarm but is under direct human control. As a result, this places those agents not under human control in a subordinate role relative to the lead agent(s), while they maintain peer relationships amongst each other. The supervisory agent becomes supervisor to the subordinate swarm members but is also subordinate to the human agent. The human supervisor can then leverage their supervisor role over one or more aircraft to indirectly control the rest of the swarm.

For example, flying a lead agent directly away from its subordinate in a 2-vehicle swarm will lead to the subordinate "chasing" the leader by following the rules governing alignment and cohesion. Flying the lead agent at a subordinate will result in the subordinate moving away from the lead vehicle due to the rules for alignment and separation. Distance spacing between aircraft will be evaluated using this type of behavior for various flight patterns in a 2-vehicle swarm and in a 3-vehicle swarm. The purpose for this is to ensure safety of flight by providing spacing guidelines to reduce the chance of midair collision.

This research is intended to be a starting point for future swarm algorithm testing by a single operator for three or more aircraft. Reynolds' rules are used as a convenient fill-in for behavior, because it's likely the separation and minimum altitude rules will be maintained for any swarm for safety purposes. The architecture behind this demonstration must be flexible enough that future iterations can replace the swarm rules with something entirely different from Reynolds' flocking rules – including a more cognitively flexible controller than a scripted algorithm.

Research was conducted in two phases: software-in-the-loop (SITL), followed by hardware-in-the-loop (HITL). At the start of this thesis, the USAF airworthiness flight release for AFIT UAV operations required one GCS per UAV. To support a release permitting one GCS to control multiple UAVs or monitor a swarm of UAVs with the option to take control, SITL testing was used to prove safety of flight. The SITL setup consisted of OSS running in multiple instances onboard a single computer; the software architecture for a single instance is shown below in Figure 3.

A FlightGear server was used to view the simulated aircraft swarm, with individual aircraft observable in their own FlightGear flight simulation instances, but also showing the other aircraft through a multiplayer server. The data was provided by JSBSim Flight Dynamics Model (FDM), receiving inputs from multiple emulated ArduPilot autopilot instances. Each autopilot was controlled separately by instances of MAVProxy GCS, a minimalist GCS capable of interfacing with Python scripts through DroneKit. A single Python script used telemetry outputs from each virtual autopilot and a swarming algorithm to provide C2 instructions. These instructions were fed through DroneKit to the MAVProxy GCS and then to the autopilots.

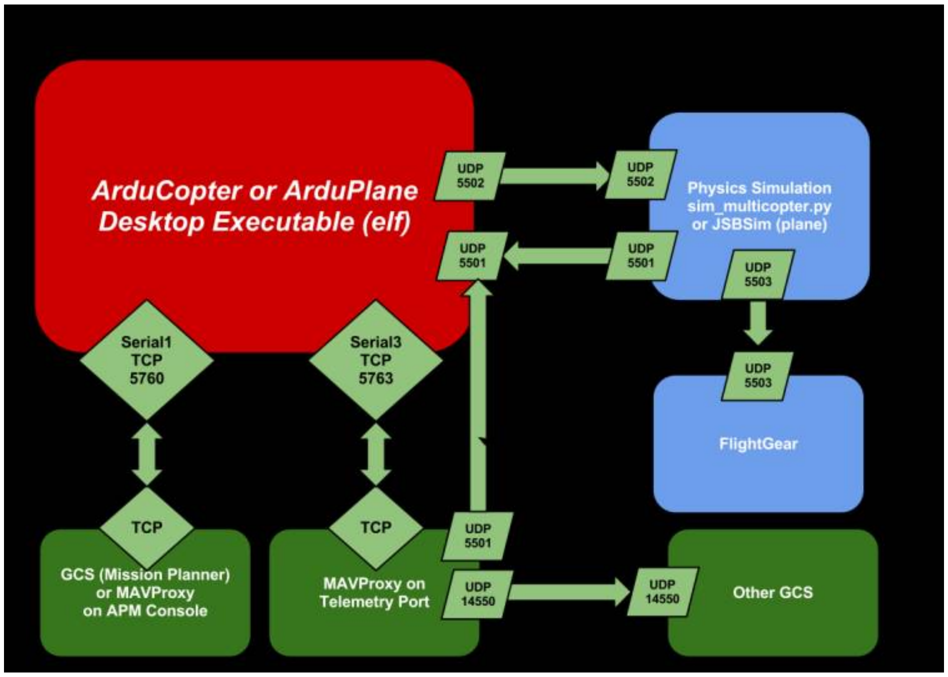

ArduCopter or ArduPlane Desktop Executable (elf)
UDP 5502
UDP 5502
Physics Simulation sim_multicopter.py or JSBSim (plane)
UDP 5501
UDP 5501
UDP 5503
UDP 5503
FlightGear
Serial1 TCP 5760
Serial3 TCP 5763
TCP
TCP
GCS (Mission Planner) or MAVProxy on APM Console
MAVProxy on Telemetry Port
UDP 5501
UDP 14550
UDP 14550
Other GCS

Figure 3. SITL Architecture (ArduPilot, 2016)

Once the swarming algorithm was shown to work in simulation, it was uploaded to actual aircraft for the second phase. Three fixed-wing UAVs were networked together, with one characterized running the same algorithm from SITL testing.

Assumptions/Limitations

Aircraft availability limits the maximum size of the swarm to three aircraft, so the scalability of the swarm architecture cannot be empirically determined. It's not feasible to simulate the limitation since the governing scripts are run on small onboard companion computers, while the simulation was run on a higher-end laptop computer. Furthermore, the type of aircraft available for flight test are limited to AFIT resources. If this architecture is successful with three aircraft, future testing can evaluate bandwidth limitations when incorporating more aircraft into the architecture.

Open-source software (OSS) will be utilized throughout this research. This provides maximum flexibility at the cost of potential security flaws (which are not addressed by this research) and reduced user-friendliness. Many commercial off-the-shelf (COTS) products will be used in this research to keep costs down. This also allows for component interchangeability for most components. The critical exception is the Pixhawk autopilot due to its ability to accept local frame velocity commands. Also, some specialized interface connections were manufactured in-house.

Implications

This research, if successful, will allow future researchers to start with a baseline of three multirotor air vehicles that exhibit a minimum level of swarming behavior in a safe and controllable manner. From there, more advanced swarming capabilities can be tested including formation flying, the addition of sensor packages, and more advanced data sharing. It will also provide early metrics for safety of flight for swarming air vehicles, and evidence in support of a reduced personnel requirement for swarms of air vehicles if those vehicles meet not-yet-established criteria for operation.

Preview

Chapter II reviews numerous publications supporting the purpose and technical background of this research. Topics include military utility of swarming aircraft, algorithms that govern autonomous flocking behavior, command and control architectures supporting swarming aircraft, methods for controlling swarm members individually and for controlling the swarm as a whole. The chapter concludes with the few documented instances where all of these components were put together and partially tested on real aircraft. Chapter III addresses the specific architecture developed by this research, the hardware and software used to test the architecture, control algorithm development for a UAV swarm, and the test and verification plan. Chapter IV discusses the results and implications of the actual tests, and Chapter V provides conclusions and recommendations for future research.

II. Literature Review

Chapter Overview

Chapter II focuses on engineering articles exploring the technical aspects of UAV swarm communication and control. Additionally, some articles delve into the military utility of UAV swarms. These articles provide the technical foundation for this thesis and show the gap it is intended to address.

Use of Low-Cost COTS Components in Multi-UAV Demonstrations

Previous research has proven the efficacy of low-cost COTS components in providing inter-vehicle communication sufficient to navigate unmanned ground vehicles (UGV) in close formation (Gray, 2015; Hardy, 2015; Toscano, 2017). Additionally, the architecture used to control two UGVs in Toscano's research, running at nearly 20Hz update rate, was assessed to be capable of including at least three more vehicles.

Military Utility

In 2002, the then-US Joint Forces Command/J9 prioritized a list of mission sets for collaborative UAV systems (USJFCOM J9, 2002). This list of missions was evaluated for specific behavioral patterns, translating militarily useful tasks into lower-level actions (Feddema, John T; Robinett & Byrne, 2004). In 2014, Kaiser adjusted the Feddema table to quantify which behaviors were common to which mission sets, shown in Table 1.

Table 1. Adjusted Feddema Behaviors (Kaiser, 2014)

Missions & Behaviors	Flocking	Converging/Diverging	Mapping/Survey	Coverage	Search	Detect/Track	Containment	Loiter	Pursuit	Attack	Evasion
Area ISR	X	X	X	X	X	X					
Point ISR	X	X		X	X	X		X			
Communication	X	X		X				X			
Navigation/Mapping	X	X	X	X		X					
Swarming Attacks	X	X		X	X	X	X		X	X	X
Defense/Protection	X	X		X		X	X	X	X	X	X
Delay/Fix/Block	X	X		X		X	X	X		X	
Deception Operations	X	X				X				X	X
(Combat) Search & Rescue	X	X		X	X	X					X

Since all the listed military missions utilize flocking and converging/diverging behaviors, it is reasonable to assume that every mission set is going to include these foundational behaviors. This thesis then, will focus on implementing those two behaviors into a scalable swarm of three or more aircraft, with sufficient remaining processing power and memory to add additional behaviors in the future.

Before proceeding, it's important to clarify the definitions of some of the terms used. In common use, flocking, herding, and swarming are all synonymous for collaborative motion, the only difference being which type of animal the behavior is describing. For the purposes of this research, flocking describes a natural-seeming aggregate behavior of individuals, which are not centrally controlled but act solely on their own perception of their environment (Reynolds, 1987). The original three Reynolds rules adequately simulate or produce flocking behavior, but lack purpose. Modifying these rules can give the aggregate behavior a purpose. Swarming for the purposes of this research is defined as a group of three or more agents acting in such a way that the behavior of each agent affects that of every other agent, and the group is collaborating to accomplish some goal which cannot be attained by an individual agent or would take an individual agent an inordinate amount of time to accomplish. Swarming does not have to include flocking behavior per se: a group of UAVs performing a gridded sensor sweep would be said to exhibit swarming behavior, but it would not appear as natural animal behavior, and thus not be described as flocking. In either case though, the positions and velocities of all other members of the swarm or flock must be available to each individual to obtain the desired effect.

Individual members of flocks or swarms, as defined, must be able to exhibit converging or diverging behavior. Converging is the ability to move towards a common point "known" by all swarm members. Diverging is the ability to move away from a common point "known" by all swarm members.

Given those definitions, the ability to flock or swarm requires two important pieces: a common coordinate frame amongst all flock members, and a way to determine each flock member's position and velocity in that frame. While it may be convenient to utilize a global frame due to common availability of satellite navigation signals, these signals can't always be relied upon in a military environment. Therefore, a local frame makes the most sense to use. When satellite navigation signals are available, the global positions can be translated to the local frame, and when not, alternative methods can be used to determine the local frame (Scaramuzza et al., 2014). Additionally, the global frame may be more useful for military missions, but the local frame may be better for flocking and converging/diverging behaviors.

The position and velocity information must be shared amongst all flock members and updated at a sufficient rate such that each member is able to respond to the movements of other members so as not to collide with them (baseline behavior) and accomplish the mission of the flock (advanced collective behavior).

Cooperative Behavior and Algorithms

In 1986, Craig Reynolds suggested an algorithm designed to simulate flocking behavior, and the following year published a paper on his findings (Reynolds, 1987). The three key rules Reynolds describes, in order of decreasing precedence, are:

Separation: avoid collisions with nearby flock members

Alignment: attempt to match velocity with nearby flock members

Cohesion: attempt to stay near other flock members

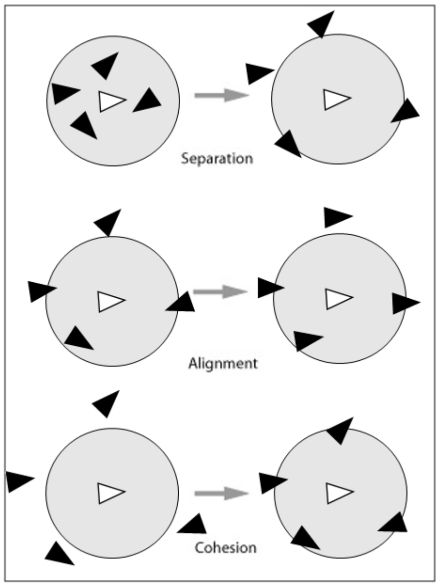


Figure 4. Original Reynolds Rules

Reynolds is careful to point out at the start of his paper that objectively measuring the success of a flocking algorithm is difficult, and notes that while some attempts to mathematically describe flocking behavior have been made, it may be more significant that many viewers are able to immediately recognize flocking behavior on sight (Reynolds, 1987). The focus of this thesis is twofold: demonstrate a functional architecture for swarm algorithm development and show safety of flight given an implementation of Reynolds' flocking rules on multirotor UAVs. Later, an objective measurement of flight safety will be attempted.

One analysis of Reynolds' original rules determines the rules as originally described lead to fragmenting behavior rather than flocking behavior (Olfati-Saber, 2006). Given random starting positions, the flock members tend to aggregate in small groups without ever forming a single large swarm. A solution is given in the form of a navigation correction: a point (which may or may not be mobile) towards which all flock members should attempt to move, whilst still obeying the original rules of separation, alignment, and cohesion. This thesis implements a hybrid solution where flock members use the geometric center of all vehicles on the network to determine the cohesion point, rather than just the geometric center of vehicles within a set radius.

Reynolds' rules have been tested in simulation for flock stability, and the specific equations governing separation, alignment, and cohesion are indeed capable of governing a stable flock. A stable flock is defined as one where all members have a common heading, stay within some defined radius of the other members, and refrain from colliding with other flock members. The potential fields governing Reynolds' rules are successfully used in both static and dynamic environments. The primary drawback to this paper is that it is purely a simulation and operates under the assumption that each flock member has near-perfect knowledge of all other flock members at all times (Tanner, Jadbabaie, & Pappas, 2003, parts I and II).

Command and Control (C2) Architectures

Based on the previous subsection, the theoretical rules governing basic swarm operation are sound and ready to be implemented in real vehicles. The information that must be shared amongst all swarm members are the local position and velocity of each member, so the next issue is how to pass that information in a complete and timely manner.

There are two overarching methods of swarm control: centralized, and decentralized, pictured below in Figure 5. Centralized control architectures rely upon ground infrastructure, limiting the capability of the swarm by tying it to a local area and increasing weight requirements (Bekmezci, Sahingoz, & Temel, 2013; Diamond, Rutherford, & Taylor, 2009). Decentralized control permits the swarm to operate out of range of ground infrastructure, and to pass data to or from any swarm member with an external link to just one swarm member (Li et al., 2008). There is a third path which combines these two methods, where one swarm member is designated the master which provides direction to the remainder. An architecture like this has been tested with some success, but at its lowest level each slave swarm member still must be able to communicate position and velocity data with the master (Pilania, Mishra, Panda, & Mishra, 2009). Therefore, a key building block of an independent swarm is a decentralized ad-hoc network.

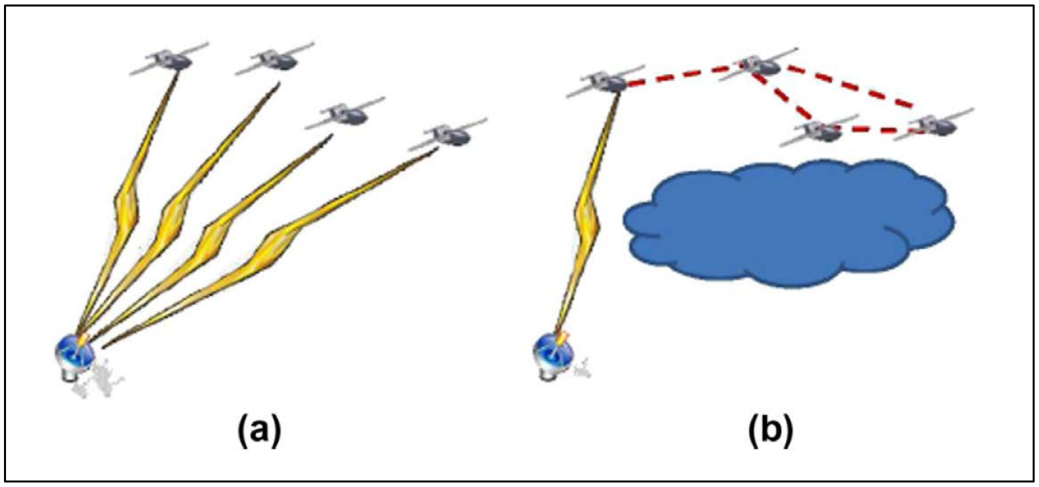


Figure 5. Centralized (a) vs Decentralized (b) Network

Having established the need for an ad-hoc network to allow swarm members to communicate amongst themselves, there are two types to choose from, displayed in Figure 6. The standard ad-hoc network assumes that each node is within communication range of every other node – a single hop for every transmission. This is not necessarily the case for a network of flying vehicles as they may be spaced over a broad area, have no hardline connecting them, and may encounter terrain obstructions. Given these circumstances, a mesh ad-hoc network, where nodes can utilize multiple hops to pass their data to other nodes, is the preferred solution (Bekmezci et al., 2013; Karl, 2005; Kumar, 2002; Li, Ming; Harris, John; Chen, Min; Mao, Shiwen; Xiao, Yang; Read, Walter; Prabhakaran, 2015; Li et al., 2008).

The Linux-based microcomputers used in previous successful multi-autonomous vehicle applications (Toscano, 2017) do have basic ad-hoc capability when paired with a Wi-Fi adapter but do not have mesh capability. Therefore, a layer-2 routing protocol is needed to fill this gap. Two open-source protocols appear to meet this requirement:

open80211s and Better Approach To Mobile Ad-hoc Networking (BATMAN) Advanced (Pojda, Wolff, Sbeiti, & Wietfeld, 2011; Shibing & Jianmei, 2016; Zafar & Khan, 2017). Both have been successfully used in mobile mesh ad-hoc networks and are suitable for UAV swarming applications (Pojda et al., 2011), although there are concerns about BATMAN-Advanced scalability (Lüssing, 2013).

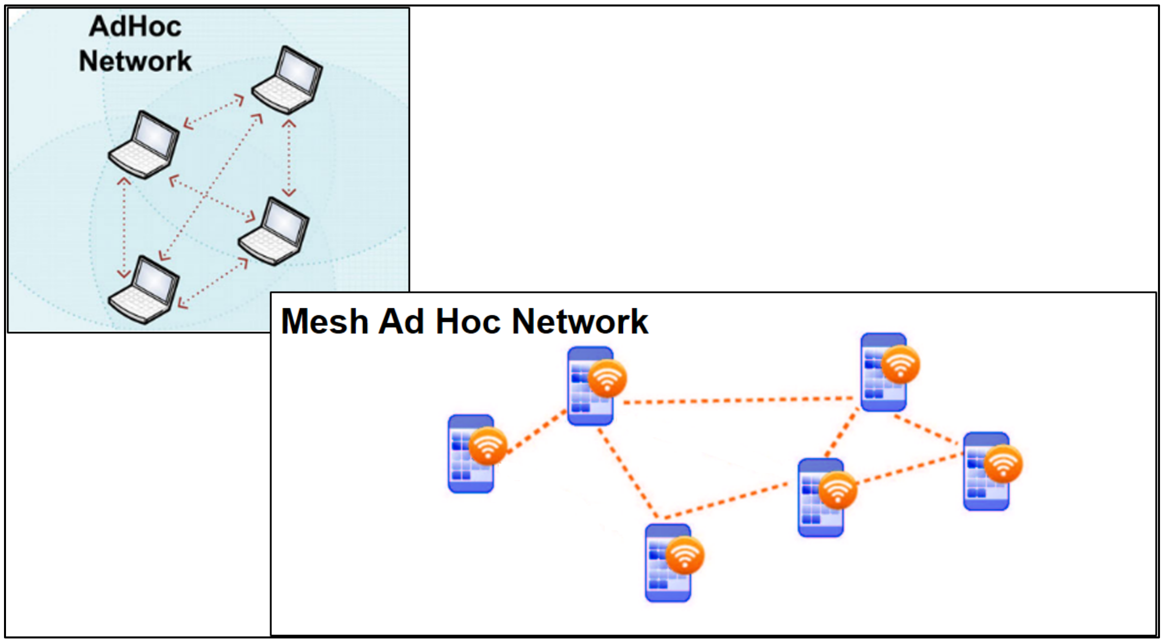


Figure 6. Ad-Hoc Network vs Mesh Ad-Hoc Network

Routing provides a network connection between vehicles, but another layer is required to pass useful data. The two common methods using Wi-Fi are TCP/IP or UDP multicast. The first method, TCP, sends messages from one node to specified nodes on the network, generating one message per node. Furthermore, it is a two-way link where receipt of the message is confirmed, and the message contents are verified. The second method is more efficient for a mesh ad-hoc network because only one message is sent out and passed until all nodes have received it. The drawback is receipt is not confirmed, and messages are not always received in order. Given that telemetry messages are only useful for a second or two at best (depending on relative location accuracy desired) and are repeated often, UDP multicast is the more efficient method to use and it also comes with unique security benefits that can be implemented (Philips, Adrian N.; Mullins, Barry E.; Raines, Richard A.; Baldwin, 2009). While there are Python modules available to send and receive multicast messages, there is a packaged solution which is inherently more flexible. The Lightweight Communications and Marshalling (LCM) library can pass different types of data structures via UDP multicast, in different programming languages, making it inherently more flexible. It has been applied to unmanned vehicles on land, air, and sea (Huang, Olson, & Moore, 2010) and is packaged in a simple, easy-to-use format, making it ideal for swarming applications.

Autonomous Control

Autonomous control of aerial swarm members is a broad topic; three facets of which will be addressed here. First, data requirements will be discussed. Then hardware and software specific to each swarm member will be considered, followed by control of the swarm as a single unit.

The common element to every swarm member regardless of the swarm’s purpose is an information requirement for position and velocity of the other swarm members around it. Methods to achieve this range from radio pulses with specialized receiver antennae (Justh, EW; Krishnaprasad, 2002), to dedicated sensors (Mcclanahan, 2017), to broadcasting telemetry (Gray, 2015; Toscano, 2017). Each method has its benefits and drawbacks, but the critical difference which makes the last option most attractive to a baseline architecture is that telemetry broadcast is independent of the data content and mode of transmission (radio or free space optical for instance). Its modularity grants flexibility, allowing other elements of the architecture to change (particularly the physical configuration of the aircraft) without changing this one. For example, local position data may be produced by vision-based sensors (Scaramuzza et al., 2014) or some other method than Global Positioning System/Inertial Navigation System (GPS/INS).

One of the drawbacks to telemetry-only swarm control is its dependency on each vehicle's sensors (GPS, INS, barometer, accelerometers) which in the case of small multirotor vehicles with commercial-quality components amounts to ~2m error in the horizontal plane and a little more on the vertical axis (Gray, 2015; Mcclanahan, 2017). However, this can be rectified by using additional sensors for localized navigation (Wilson, Ali, & Sukkarieh, 2015), or real-time kinematic (RTK) differential GPS solutions (McCollum, 2017). There is also some latency between the autopilot processing the data from its positional sensors and the receipt of that data by other swarm members. Methods are available to reduce this issue (Woolley, Peterson, & Kresge, 2011) but are not investigated further.

At the hardware level, small UAVs require the use of an onboard autopilot to maintain flight stability. Open-source autopilots, specifically the 3D Robotics Pixhawk, have been used successfully in two-vehicle teams in a leader-follower arrangement (Gray, 2015; Toscano, 2017), with indications that it is suitable to provide control for three or more vehicles in a swarm. These autopilots do not include the ability to execute specialized scripts or communicate remotely to other autopilots, however this is addressed by the use of a small companion computer – often a Raspberry Pi or Beaglebone Black (Toscano, 2017). Ardu-based autopilots can interface on a software level with these companion computers through the open-source modules contained in DroneKit-Python, but physical connections can be tricky (Toscano, 2017). Companion computers can then use virtually any type of radio to transmit and receive data externally to the air vehicle. The proprietary Wave Relay system has been shown to be effective (Gray, 2015; Toscano, 2017), but ad-hoc Wi-Fi networks are also capable depending on the layer-2 protocol used (Bekmezci et al., 2013; Zafar & Khan, 2017).

Having addressed control issues for individual swarm members, it’s also important to consider control of the swarm as an entity of its own. The small UAV flight release used by AFIT currently requires a crew of three people per aircraft – an operator, safety pilot, and observer. One of the goals of this thesis is to provide evidence of swarm safety of flight in an effort reduce manpower required to operate a swarm of UAVs. The US Federal Aviation Administration (FAA) notes no formal testing process has been developed for swarming technology by any nationally-recognized organizations (Federal Aviation Administration; Office of the Secretary of Transportation; Department of Transportation, 2016). The FAA is aware that swarming is a desirable technology for both civilian and military use however, and is working to create rules allowing one pilot to control multiple coordinated vehicles (Duncan, 2017). Research building off Reynolds’ and Olfati-Saber suggests that simple manual swarm control can be achieved with just a single lead vehicle which broadcasts its position and velocity as if part of the swarm but whose movements are not restricted to Reynolds’ rules (Su, Wang, & Lin, 2007). Existing research is theoretical; this thesis will demonstrate a functional swarm where one aircraft is controlled manually as the leader, while the swarming follower vehicles attempt to match its velocity, subject to Reynolds' rules.

Combining Behavior Algorithms, C2 Architecture, and Autonomous Control

There are two excellent examples implementing Reynolds' rules in an actual drone swarm that are available for study. The first was documented in 2011 and involved ten fixed-wing aircraft (Hauert et al., 2011). As this thesis uses multirotor vehicles, there are fewer constraints since the multirotor aircraft do not have to maintain forward velocity in the body frame to stay aloft. The second example implemented Reynolds' rules in multirotor aircraft, with the additional ability to replace a strict cohesion rule with a formation-cohesion rule (Vasarhelyi et al., 2014). Formations included a ring, grid, or line as desired, and maneuvered the center of gravity of the swarm (and thus the swarm) while still maintaining formation. In both examples, each swarm member was forced to maintain a specific altitude to avoid collision, and all maneuvering per Reynolds' rules was conducted in the horizontal plane. This thesis will demonstrate a swarm of three multirotor aircraft operating fully in three dimensions rather than just two.

Conclusion

Based on available literature, elements of a UAV swarm architecture have been theorized, simulated, and tested. In some cases, they have been assembled and tested as whole system with nominal two-dimensional limitations. However, no comprehensive and tested architecture for a scalable swarm of three or more air vehicles has been publicly documented. This thesis will start to fill that gap and is intended to be a starting point for real-world testing of mission-based swarming algorithms built on a common baseline.

III. Methodology

Chapter Overview

The purpose of this chapter is to describe a testable baseline architecture for a UAV swarm of three or more vehicles and the methods used to test that architecture. First the architecture is described, followed by the hardware and software implementation. Next, the control algorithm is discussed along with software test techniques, and the chapter concludes with the test and verification procedure for the UAV swarm.

Architecture

This system architecture will be defined using views from the Department of Defense Architecture Framework (DODAF). In keeping with best practices, the architectural views utilized will be those sufficient and appropriate to describe the system.

The following pair of operational concept graphics (OV-1) in Figures 7 and 8 depict first a typical scenario used to test the architecture in this research, and second a hypothetical scenario that this architecture would be able to achieve if the swarm had an autonomous objective – which is only tested to a small extent in this research.

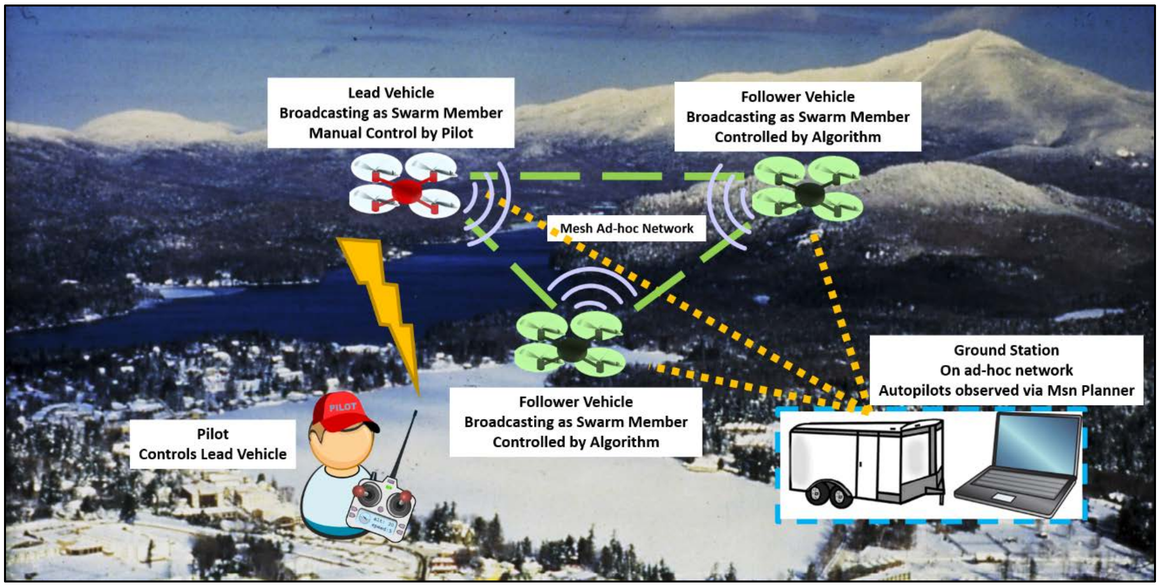


Figure 7. OV-1, Tested Swarm Configuration

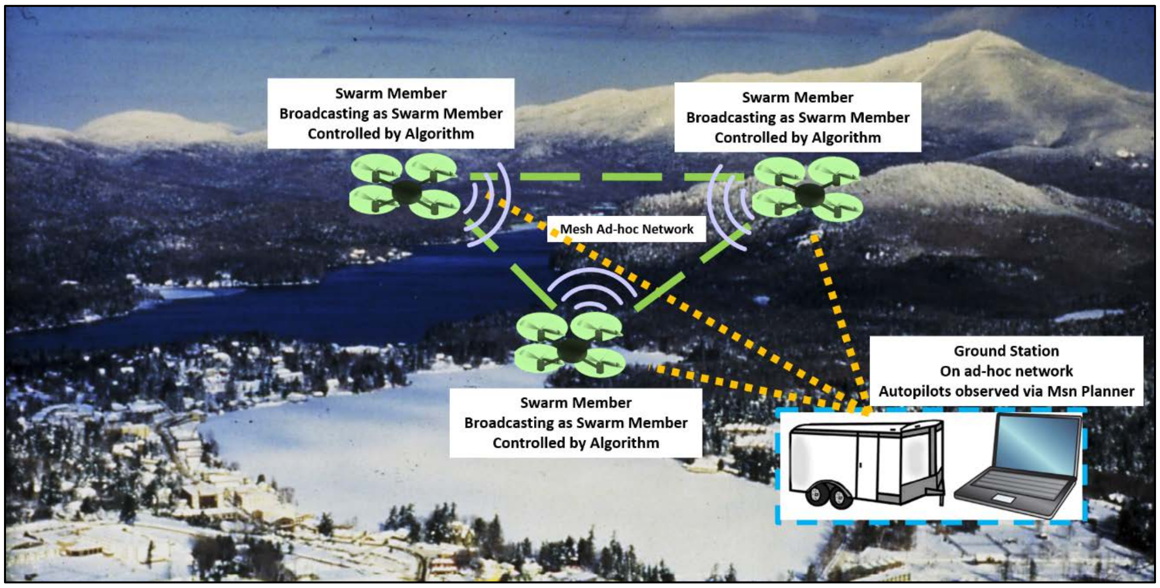


Figure 8. OV-1, Notional Swarming Configuration

The following pair of abbreviated use cases apply to the OV-1 from Figure 7. The full use cases and definitions are provided in Appendix A. An activity diagram is also provided below each use case. Note the behaviors associated with the activity diagrams are assigned to generic vehicles; however, in the use cases they are specifically assigned to multirotor aircraft.

Use Case Example 1

Pre-Conditions: *The X-8 multirotors are airborne in altitude hold mode, spaced approximately 30-50 meters apart, at an altitude of 20-30 meters AGL.*

Main Flow:

1. The non-lead safety pilots change the mode of their vehicles from altitude hold to guided.
2. The non-lead X-8 multirotors fly autonomously towards the geometric center of the swarm.
3. Once the X-8 multirotors are within 20 meters of each other, their motion changes as the alignment vectors cancel out.
4. The vehicles slow as they approach the center and start to fly apart if they are too close to any other vehicles.
5. This behavior is permitted to continue for a minute or two for sufficient data collection.

Post-Conditions: *The X-8 multirotor vehicles achieve a dynamic equilibrium near the lead vehicle, continuously attempting to move towards the geometric center of the swarm, then repelled by the proximity of other vehicles. Data is collected for a minute or two to characterize the behavior.*

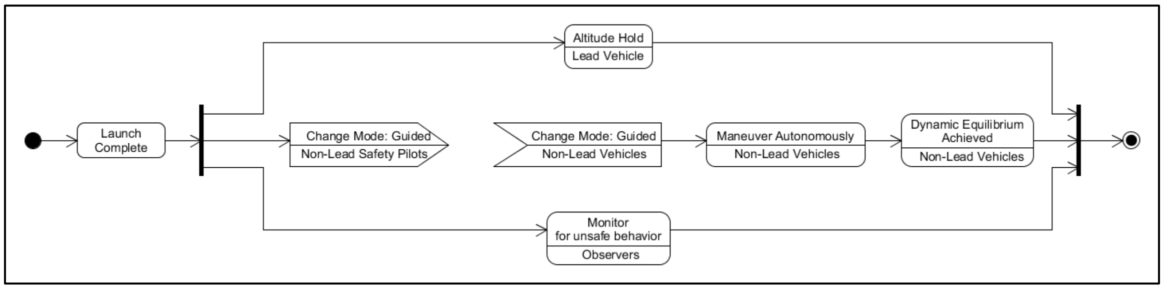


Figure 9. Activity Diagram, Use Case 1

Use Case Example 2

Pre-Conditions: *The X-8 multirotor vehicles have achieved a dynamic equilibrium near the lead vehicle, continuously attempting to move towards the geometric center of the swarm, then repelled by the proximity of other vehicles, and data has been collected.*

Main Flow

1. The lead safety pilot maneuvers the lead X-8 multirotor away from the other aircraft.
2. The non-lead X-8 multirotor vehicles autonomously maneuver to follow the lead vehicle.
3. The lead safety pilot maneuvers the lead X-8 multirotor in benign patterns – straight lines or gentle arcs.
4. The non-lead X-8 multirotor vehicles maneuver autonomously in response.
5. This maneuvering is done for a minute or two for sufficient data collection.

Post-Conditions: *The non-lead X-8 multirotor vehicles maneuver in response to the manually-controlled lead vehicle, according to the Reynolds+ algorithm rules. Data is collected for a minute or two to characterize the behavior.*

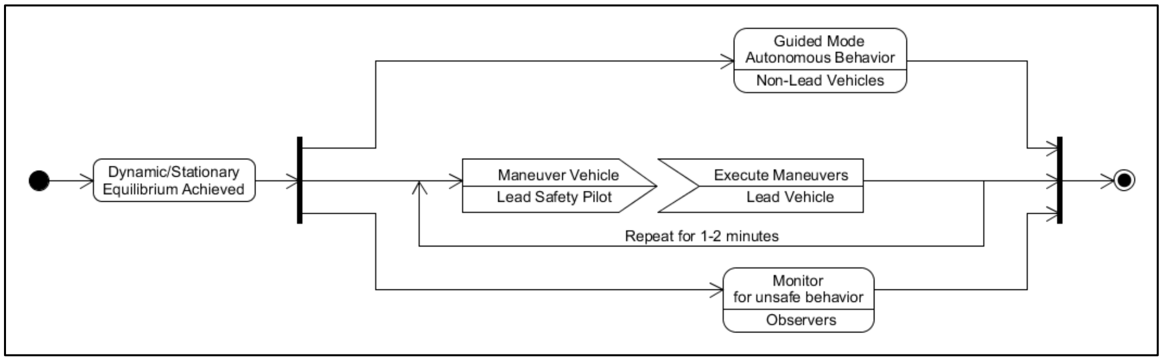


Figure 10. Activity Diagram, Use Case 2

Since the architectural vision and overview has been described, the architecture will now be shown in greater detail. First, the OV-5b (Figure 11) maps operational activities to the system components responsible for activities and documents how those activities will flow during normal operations. Note that while each aircraft is capable of manual or guided flight, tests during this research will always have at least one aircraft under manual control.

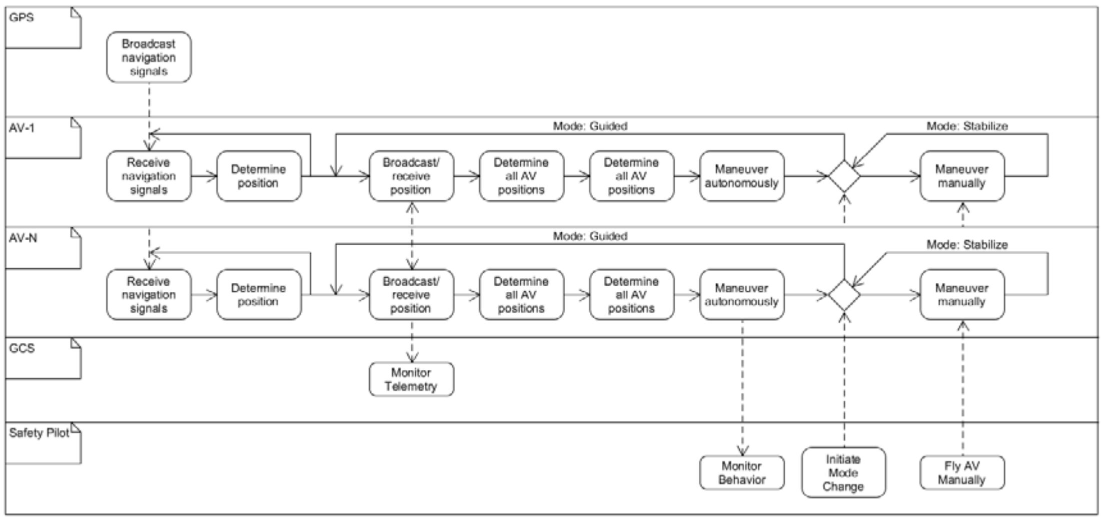


Figure 11. Activity Diagram with allocated swim lanes

The next viewpoint (Table 2) is a fit-for-purpose view, useful in mapping the architecture's physical layers, and translating the operational activities (OV-5b) to subsystem functions. This is similar to a SV-5b but provides more details. Function descriptions are provided in Appendix B. Note a critical capability of the autopilot is it must be able to accept velocity commands in a local frame of reference.

Table 2. Layer-Component-Function Table

Layer 1	**Swarm System**	**Components/Subsystems**	**Objective**
		Air Vehicle Swarm	Accomplish Generic Mission Fly Navigate/Operate Safely
		Ground Control Station	
		Global Positioning System	
		Safety Pilots	
Layer 2	**System**	**Subsystem**	**Functions**
	AV Swarm	Air Vehicle	Fly Navigate Safely Maneuver as a group
	Safety Pilot	Human	Mode changes Fly vehicle manually Monitor AV behavior Manual recovery as needed
		Radio Control	
	GPS	Space Vehicles	Provide nav Signals
	GCS	Human	Monitor autopilot telemetry
		Laptop/PC	
		Radio Transceiver	
Layer 3	Air Vehicle	Multirotor aircraft	Comm w/GCS Receive nav signals Comm w/each other Fly autonomously Fly manually Receive safety pilot cmds
		Strap-on guidance package	
Layer 4	Strap-on guidance package	Companion Computer	Comm w/other guidance packs Get telemetry from aircraft Send velocity cmds to aircraft
		WiFi Adapter	
		Battery	
	Multirotor aircraft	Frame	Fly manually Fly autonomously Comm w/safety pilot radio Comm w/GCS Send telemetry to guidance pkg Rcv vel cmds from guidance pkg Rcv GPS signals Determine position
		Motors	
		Props	
		Speed Controllers	
		Battery	
		Autopilot + GPS Receiver	
		Remote Ctrl Radio	
		GCS Radio	
		GPS Antenna	
Layer 5	Autopilot + GPS Receiver		Rcv GPS signals Determine position Send telemetry to guidance pkg Rcv vel cmds from guidance pkg Send PWM signals to motors Send/rcv signals - GCS radio Send/rcv signals - safety pilot

With the functions allocated to subsystems, the interfaces between those systems can now be described in a pair of system interface views (SV-1). The first SV-1 (Figure 12) shows the interfaces at layers two and three (as described in Table 3), between the UAV swarm itself and all external connections: the safety pilots, the ground control stations (GCS), the Global Positioning System (GPS), and between the air vehicles. The second SV-1 (figure 13) shows the interfaces at layers three and four: between the guidance package and the air vehicle itself, and again between the air vehicles.

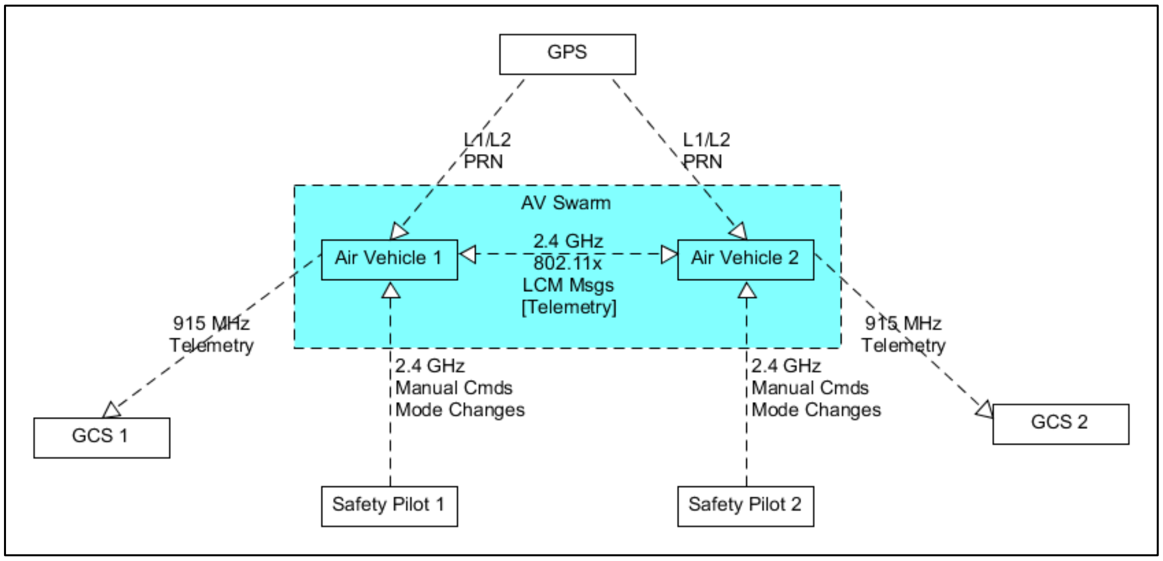


Figure 12. System Interface Description (SV-1), Layers 2-3

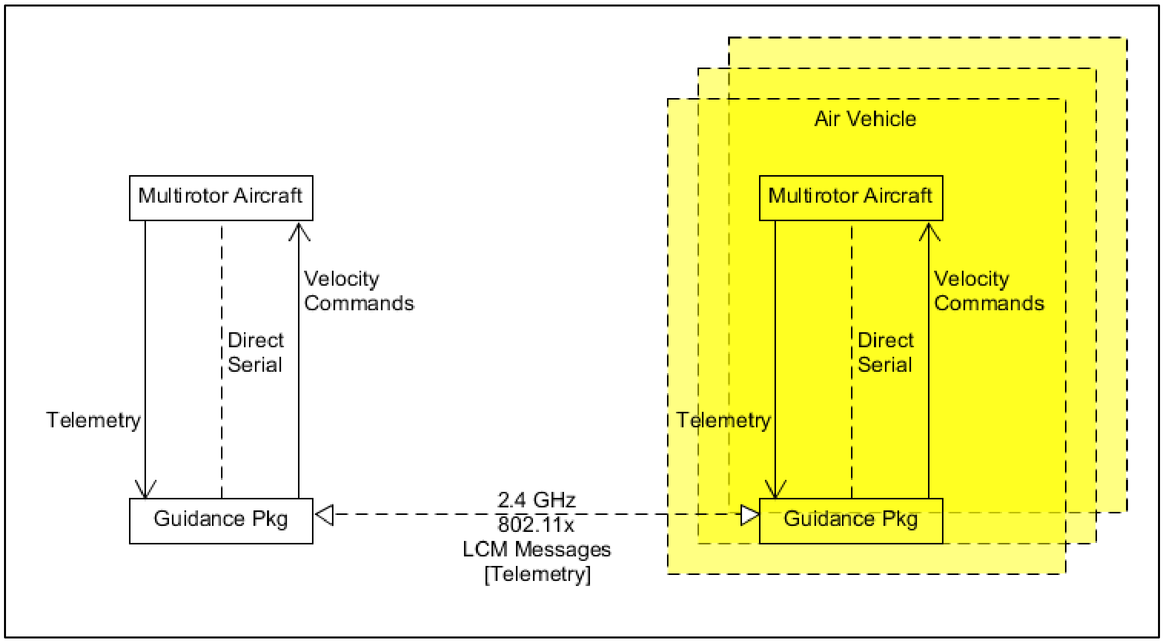


Figure 13. System Interface Description (SV-1), Layers 3 and 4

The architecture has been sufficiently described through the above DODAF views to implement it in hardware and software. Note that many of the views above contain specific radio frequencies; these are not necessarily dictated by the architecture but due to FCC restrictions they are bands open for the purposes assigned and are commonly used in small UAV applications.

Hardware/Software Implementation

Before discussing the hardware and software choices at length, it's important to discuss the values behind component selection for this research. The architecture described previously allows for a great deal of flexibility in component selection. The availability of institutional resources puts some limitations on components. Where choices were open to virtually any available COTS or open source product, the specific components selected were chosen for functionality, ease of implementation, flexibility, and low cost.

Three X-8 multirotor aircraft (figure 14) were provided for research and test. Each X-8 has eight rotors arranged in an X-pattern with one blade on the top and one on the bottom at the end of each arm. These aircraft have been used previously in single and multi-UAV tests and provide more than adequate performance up to 10 m/s horizontally and 5 m/s vertically in winds up to 20 kph. The architecture and software used could just as easily be used with any multirotor airframe. The only critical piece of hardware in this setup is the 3D Robotics Pixhawk autopilot, which is capable of receiving velocity commands in a North-East-Down (NED) frame. Not every autopilot, open-source or otherwise, has this capability but it is required within the architecture. The X-8s each include two radios: one 915 MHz radio for connecting to a GCS, and one 2.4 GHz radio for manual control by the safety pilot.

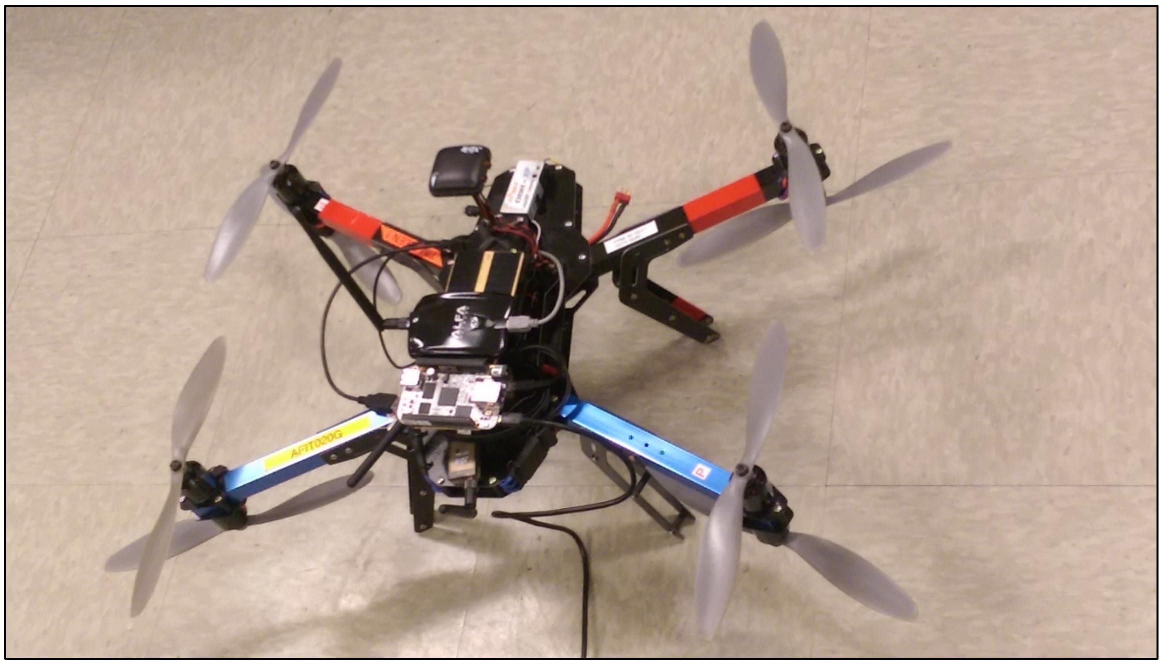


Figure 14. X-8 Multirotor with Strap-on Guidance Package

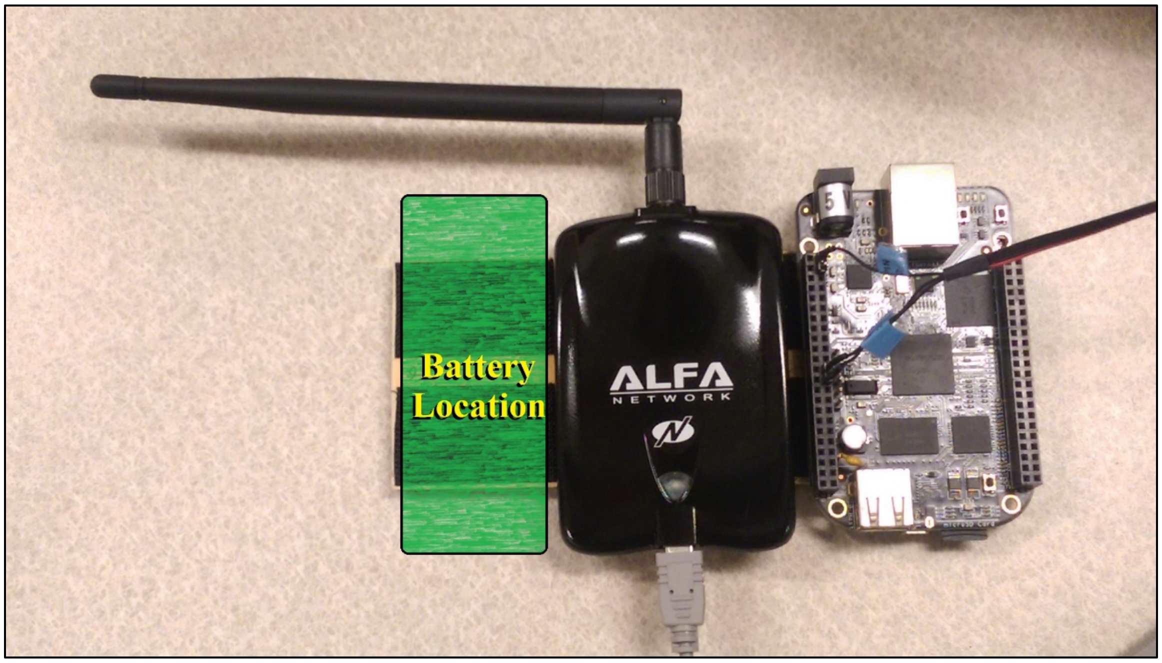


Figure 15. Strap-on guidance package (battery location outlined)

The strap-on guidance package (Figure 15) consists of a Beaglebone Black companion computer, an Alfa AWUS036NHA Wi-Fi adapter, a two-cell 2200 mAH Lithium-Polymer (LiPo) battery, and a voltage regulator. An optional colored LED array was also purchased to aid ground personnel in distinguishing each platform. This guidance package is attached via hook-and-loop to the multirotor, and interfaces directly with the Pixhawk autopilot through a custom serial cable. The serial cable connects the UART1 port on the Beaglebone Black to the Telem2 port on the Pixhawk. The block diagram in Figure 16 shows the physical elements comprising the air vehicle, the various interfaces and data links, and the information that flows between components.

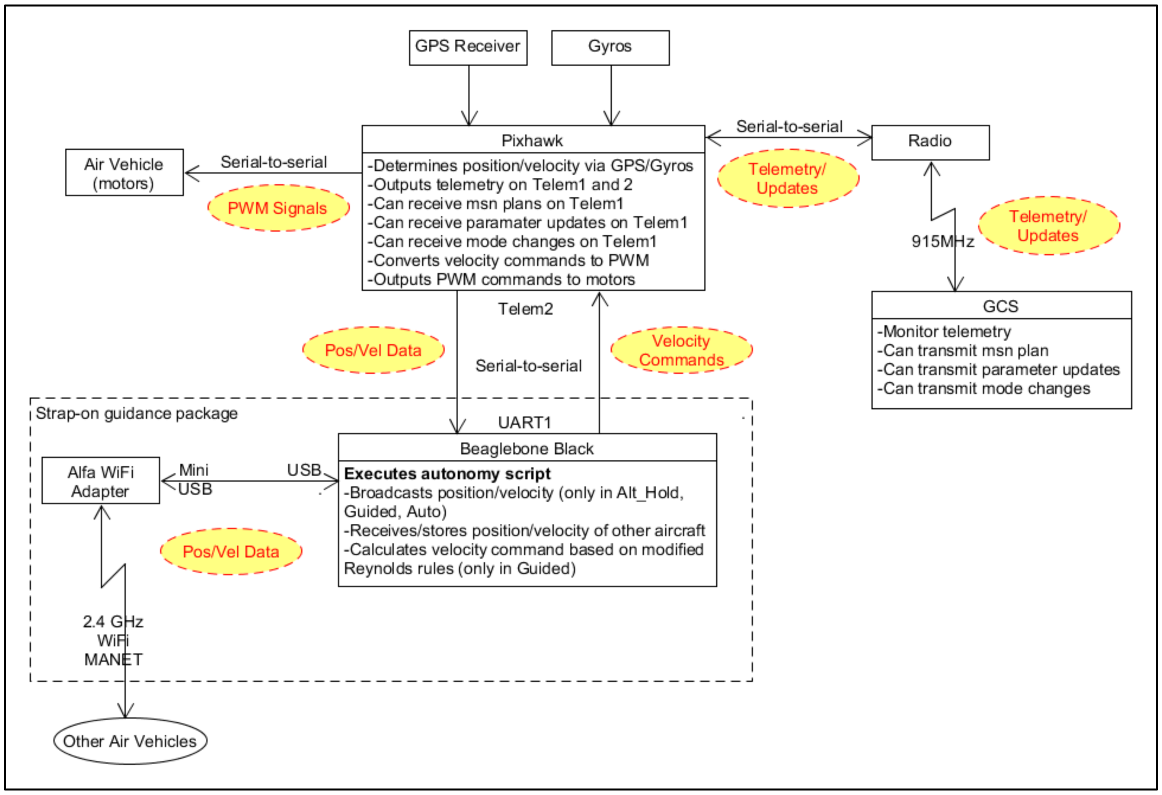

GPS Receiver
Gyros
Pixhawk
-Determines position/velocity via GPS/Gyros
-Outputs telemetry on Telem1 and 2
-Can receive msn plans on Telem1
-Can receive paramater updates on Telem1
-Can receive mode changes on Telem1
-Converts velocity commands to PWM
-Outputs PWM commands to motors
Serial-to-serial
Radio
Telemetry/ Updates
Air Vehicle (motors)
Serial-to-serial
PWM Signals
915MHz
Telemetry/ Updates
GCS
-Monitor telemetry
-Can transmit msn plan
-Can transmit parameter updates
-Can transmit mode changes
Telem2
Pos/Vel Data
Serial-to-serial
Velocity Commands
Strap-on guidance package
UART1
Beaglebone Black
Executes autonomy script
-Broadcasts position/velocity (only in Alt_Hold, Guided, Auto)
-Receives/stores position/velocity of other aircraft
-Calculates velocity command based on modified Reynolds rules (only in Guided)
Alfa WiFi Adapter
Mini USB
USB
Pos/Vel Data
2.4 GHz WiFi MANET
Other Air Vehicles

Figure 16. Block Diagram

Regarding the software, the Pixhawk autopilot is running APM: Copter version 3.4.6. The companion computer is running a Debian Operating System (OS) but any Linux distribution will suffice. The connection to the Pixhawk from the companion computer is made through DroneKit-Python, so all autonomy scripting is also written in Python. BATMAN-Advanced is used to set up the mesh ad-hoc network using the Alfa Wi-Fi adapters, and a script was written to automatically connect to the network upon powering the companion computer. The Lightweight Communications and Marshalling (LCM) library was chosen to facilitate UDP multicast over the network due to its flexible nature and also for data collection as it contains native data-logging capability (Huang et al., 2010).

Algorithm Development and Verification

The autonomy algorithm and software verification method will be described next. Although the software test environment was completed before the control algorithm chronologically, the control algorithm will be described first and then the software test will be addressed.

Reynolds+ Rules

As mentioned, Reynolds' rules of separation, alignment, and cohesion provide a foundation for swarming behavior. In a computer simulation, the velocity calculated by the rules is simply added to the current position to provide the next position of the flock member. On real aircraft, a desired velocity in the local NED frame is calculated during one loop of the control cycle, and then that velocity is sent as a command to the autopilot, which attempts to match it until another command is sent, or the first command expires. For the Pixhawk, velocity commands expire after a maximum of one second, so the control loop must be at a higher frequency. Additional factors come into play when implementing these rules on small UAVs that aren't present in simulation. In addition to separation between vehicles, it is prudent to avoid ground collision as well, so a fourth rule was adopted to enforce a minimum altitude (flight deck) using a potential field like the separation rule. A fifth rule requiring aircraft to remain within a specified radius of the GCS to retain ground control of the vehicle if necessary was considered but not implemented in this research. The modified Reynolds rules are shown in Figure 17, alongside the original rules. The combination of the original rules and the new ones are coined "Reynolds+."

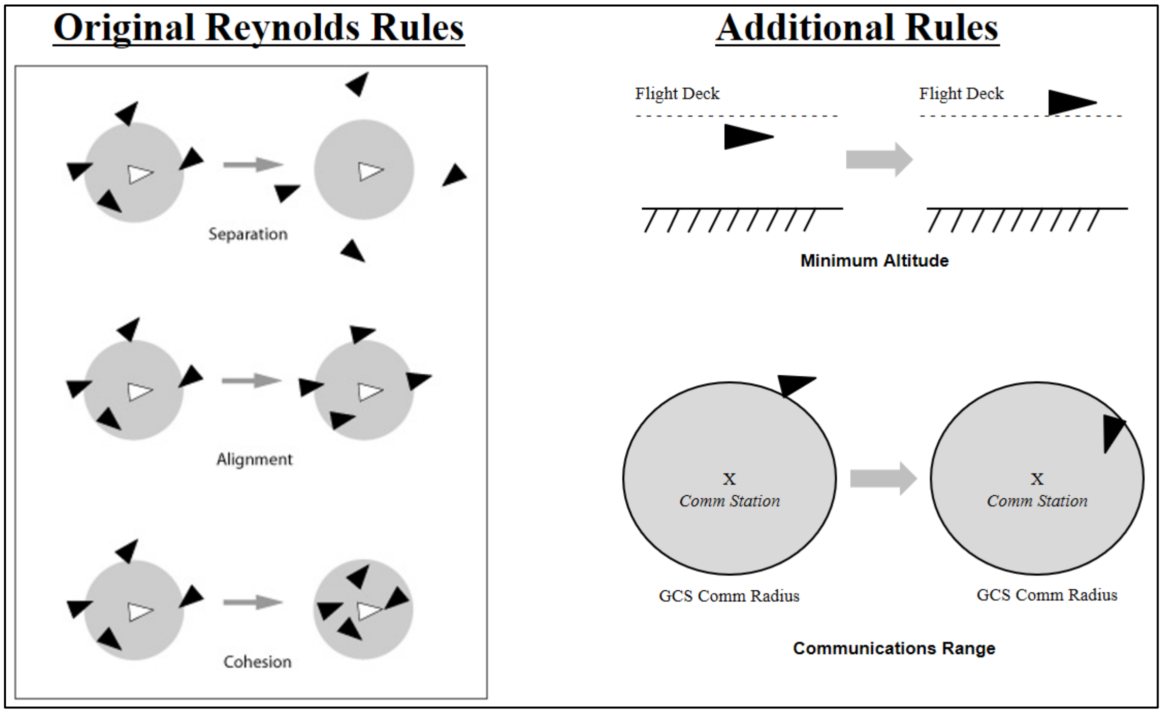


Figure 17. Reynolds+ Rules

The telemetry received from other aircraft is combined with the aircraft's own data pulled from the autopilot by the companion computer. As the aircraft's own data is retrieved in global coordinates, it is then translated into a common local tangent plane (LTP) used by all aircraft in the swarm (see Figure 18 and Equations 1-3)(Drake, 2002). From there it is simultaneously used in the Reynolds+ calculation and sent out to other swarm members to use. The resulting velocity after all rules are accounted for is then transmitted from the companion computer to the autopilot.

$$\boldsymbol{X = (N(\phi) + h)cos\phi cos\lambda}$$

$$\boldsymbol{Y = (N(\phi) + h)cos\phi sin\lambda} \qquad (1)$$

$$Z = (N(\phi)(1 - e^2) + h)sin\phi$$

where:

X = axis from Earth's center of mass to equator and prime meridian intersection
Y = 90° counterclockwise (from north) offset from X-axis along the equator
Z = axis from Earth's center of mass to the north pole
N = see Eqn. 2
h = height above ellipsoid
ϕ = latitude (geodetic)
λ = longitude (geodetic)

$$\boldsymbol{N(\phi) = \frac{a}{\sqrt{(1 - e^2 \sin^2 \phi)}}} \qquad (2)$$

where:

a = semi-major axis
e = ellipsoid first numerical eccentricity

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} -sin\lambda_{ref} & cos\lambda_{ref} & 0 \\ -sin\phi_{ref}cos\lambda_{ref} & -sin\phi_{ref}sin\lambda_{ref} & cos\phi_{ref} \\ cos\phi_{ref}cos\lambda_{ref} & cos\phi_{ref}sin\lambda_{ref} & sin\phi_{ref} \end{bmatrix} \begin{bmatrix} X_{veh} - X_{ref} \\ Y_{veh} - Y_{ref} \\ Z_{veh} - Z_{ref} \end{bmatrix} \qquad (3)$$

where:

ϕ_{ref} = reference point latitude (geodetic)
λ_{ref} = reference point longitude (geodetic)
X_{veh} = vehicle X-coordinate (ECEF)
Y_{veh} = vehicle Y-coordinate (ECEF)
Z_{veh} = vehicle Z-coordinate (ECEF)
X_{ref} = reference point X-coordinate (ECEF)
Y_{ref} = reference point Y-coordinate (ECEF)
Z_{ref} = reference point Z-coordinate (ECEF)
x = vehicle east-coordinate (LTP)
y = vehicle north-coordinate (LTP)
z = vehicle up-coordinate (LTP)

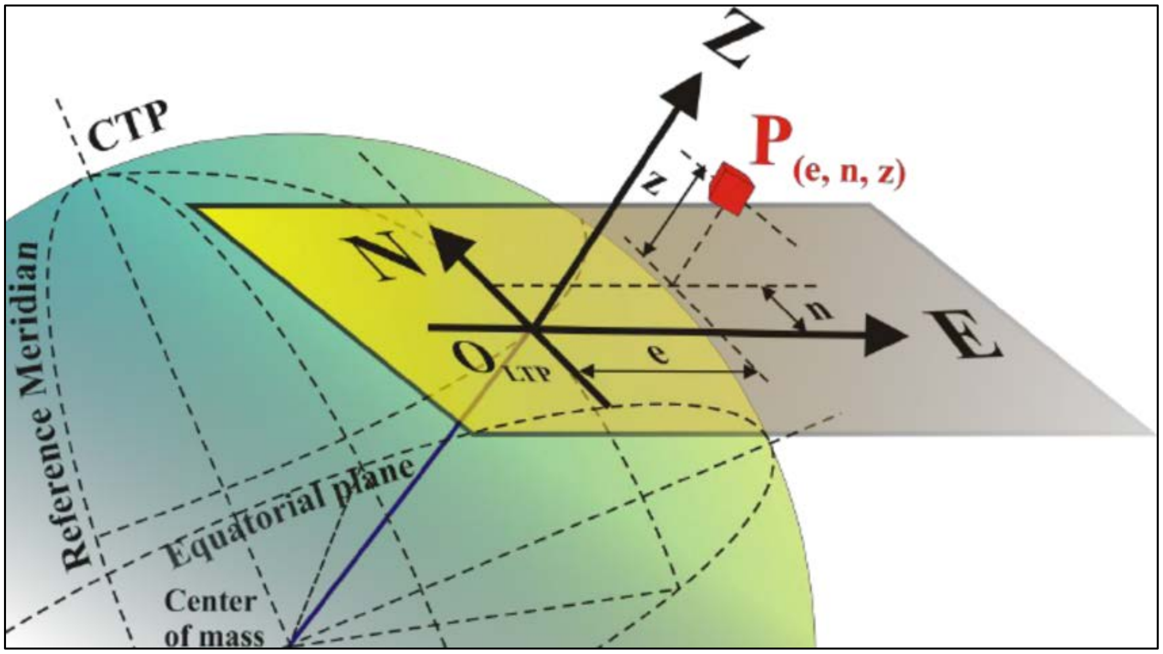


Figure 18. Global Frame to Local Tangent Plane Conversion

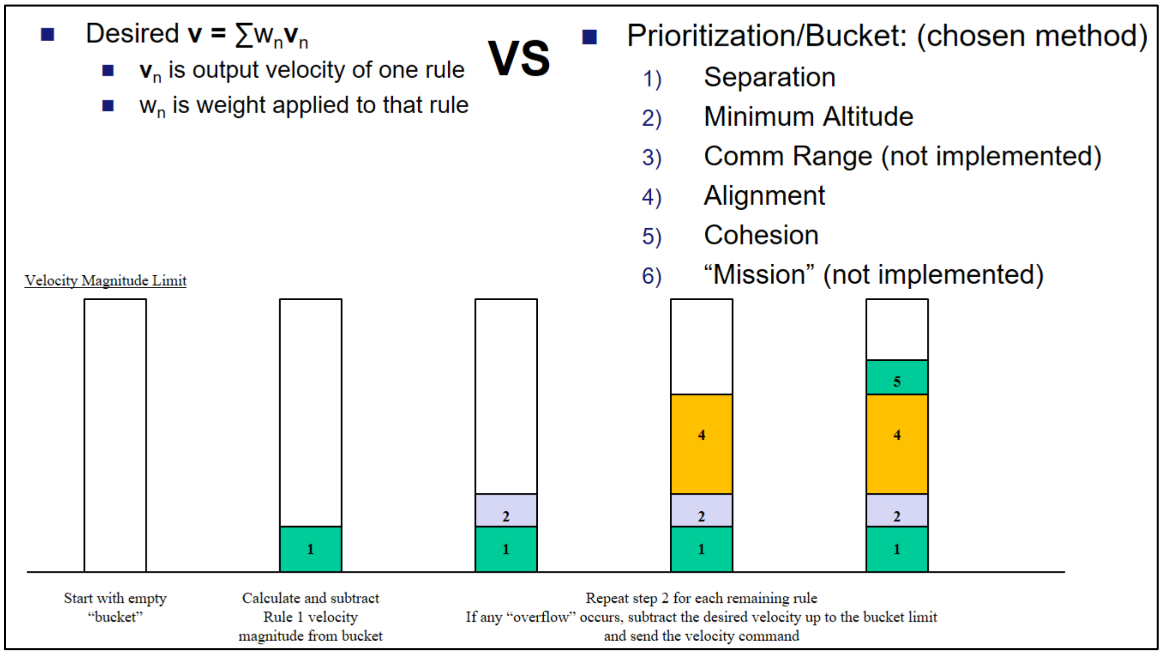
Bucket Method for Velocity Calculations

All documented implementations of Reynolds' rules use a weighted sum method to calculate the desired velocity of each swarm member. That is, the target velocity for each rule is multiplied by some factor depending on the desired importance of that rule, and the resulting velocities are summed into the swarm member's desired velocity for each control loop. This has the potential to result in undesirable behavior because the velocity for a lower-priority rule might grow high enough to overcome its low multiplier and result in a collision. Instead, the author prioritized the rules and set a velocity magnitude limit, or bucket. Reynolds proposed a similar system in his original paper but focused on acceleration rather than a target velocity (Reynolds, 1987). Each rule's magnitude is added to the bucket in priority order, and once the bucket is "full," the remaining rules are discarded along with excess magnitude from the rule that filled the bucket. These two methods are shown in Figure 19, and the bucket method is described in a graphic sequence. Another benefit of the bucket method is rules can be added and discarded with ease instead of having to recalculate a weighted system every time a change is made.

$$\vec{u}_j = \sum_{r=1}^{|rules|} w_r \vec{u}_{r,j} \qquad (4)$$

where:

$\vec{u}_{r,j}$ = velocity commanded by rth rule to the jth vehicle
w_r = weight applied to rth rule



Desired **v** = ∑w$_n$**v**$_n$
v$_n$ is output velocity of one rule
w$_n$ is weight applied to that rule
VS
Prioritization/Bucket: (chosen method)
1) Separation
2) Minimum Altitude
3) Comm Range (not implemented)
4) Alignment
5) Cohesion
6) "Mission" (not implemented)
Velocity Magnitude Limit
1
2
4
5
Start with empty "bucket"
Calculate and subtract Rule 1 velocity magnitude from bucket
Repeat step 2 for each remaining rule
If any "overflow" occurs, subtract the desired velocity up to the bucket limit and send the velocity command

Figure 19. Weighted-Sum Velocity vs Velocity Magnitude Bucket

Rule Prioritization and Calculation

The governing rules depicted in Figure 19 were prioritized to value safety of flight over mission. Separation, minimum altitude, and communications range are all safety-related, while alignment, cohesion, and mission execution are task-related. From there, the rules were prioritized in order of most likely occurrence based on the test plan shown later in this chapter. The air vehicles will all be operating within visual range of the safety pilots and ground control for this research, so communications radius was given the lowest priority and not actually implemented. Nominal flight altitude for all tests was designated to be in the 20-30m range, so even if a swarm member were directly below a lead aircraft or the swarm center, it would still be operating at an altitude of 10-20m. That allows time for the safety pilot to recover the vehicle safely if needed. Therefore, inter-vehicle collisions are the most likely safety issue with this research, so vehicle separation was given the highest priority.

Note that the equations as presented show some specific numerical values. These values were selected by SITL test trial-and-error. A heuristic provided starting values: plots of behavioral responses were sketched, with cohesion and separation becoming equal at the desired separation radius, and a steeper slope for separation than cohesion. For behaviors which only appear at a certain distance from something (another vehicle or the ground), the magnitudes were started near-zero, and exponentially increased to provide a smooth response. Over the course of testing the equations were changed to provide smooth behavior as viewed in the SITL environment. They are not "optimized" for any specific behavior patterns and can be adjusted to, for instance, provide swifter response to an encroaching vehicle within the separation radius.

$$\vec{\boldsymbol{u}}_{\boldsymbol{1},\boldsymbol{j}} = \sum_{\boldsymbol{r}=\boldsymbol{1}}^{|\boldsymbol{intruders}|} \vec{\boldsymbol{v}}_{\boldsymbol{r},\boldsymbol{j}}\left(\frac{\boldsymbol{100}}{\boldsymbol{d}_{\boldsymbol{r},\boldsymbol{j}}+\boldsymbol{7}} - \boldsymbol{5.7}\right) \tag{5}$$

where:

> |intruders| = total number of vehicles within separation bubble radius
> $\vec{v}_{r,j}$ = unit vector from *r*th encroaching vehicle to the *j*th vehicle (NED frame)
> $d_{r,j}$ = distance (meters) from the *r*th encroaching vehicle to the *j*th vehicle

$$\vec{u}_{2,j} = \vec{v}_2\left(\frac{1000}{\left(h_j + 5\right)^2} - 5.7\right) \qquad (6)$$

where:

> $\vec{v}_2$ = [0 0 -1]
> h_j = height of *j*th vehicle above ground (meters)

$$\vec{u}_{4,j} = \frac{\sum_{r=1}^{|numVeh|} v_r}{numVeh} \qquad (7)$$

where:

> numVeh = number of vehicles within the designated alignment radius
> v_r = velocity of *r*th vehicle within the alignment radius (m/s)

$$\vec{u}_{5,j} = \vec{v}_{5,j}\left(\frac{5d_{5,j}}{42} - \frac{20}{21}\right) \qquad (8)$$

where:

> $\vec{v}_{5,j}$ = unit vector from *j*th vehicle to the geometric center of all vehicles (NED)
> $d_{5,j}$ = distance to the geometric center of all vehicles (meters)

Each rule generates a target velocity (m/s) for the aircraft, which in every case but one is a function of distance from the air vehicle to a specific point. The only exception is alignment, which is an averaged velocity of all nearby air vehicles. These rules were implemented in up to three different ways: once for testing inside a caged environment, once for testing in an unenclosed space with wide spacing (for safety of flight), and last for testing unenclosed with narrow spacing (originally used for software testing). Each rule also has limitations on when it comes into effect. Plots of rules 1, 2 and 4 as implemented are shown in figures 20 and 21, and equations 5-8 show the velocity calculations for rules 1, 2, 4, and 5 respectively; for more information see Appendix C.

Telemetry Transmission and Logging

LCM is used to broadcast telemetry over the ad-hoc network, as well as the velocity commands sent to each autopilot. Only the current position and velocity information is needed for swarming capability, but both are recorded to provide redundant data records of all maneuvers. Each aircraft records all the outbound and inbound LCM messages automatically, so in theory each aircraft has a full record of the swarm's movements. Telemetry is also recorded at a higher rate on-board the Pixhawk autopilots.

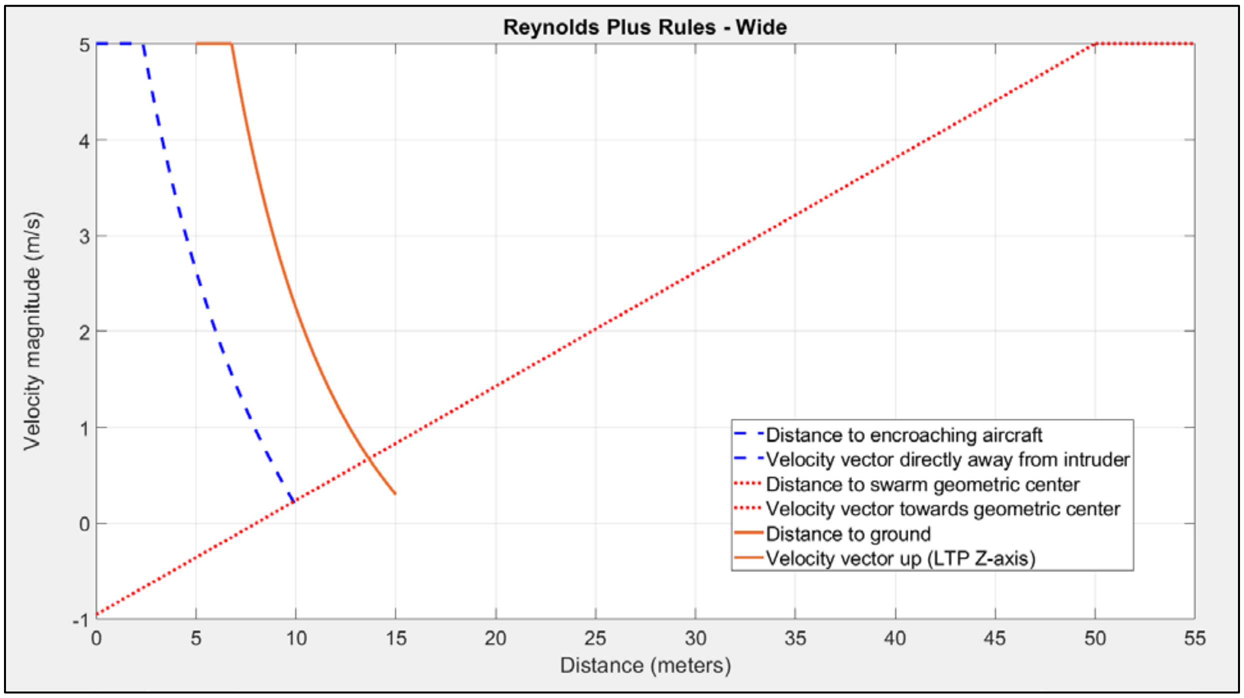


Figure 20. Reynolds+ Rule Magnitude Plots, Wide Version

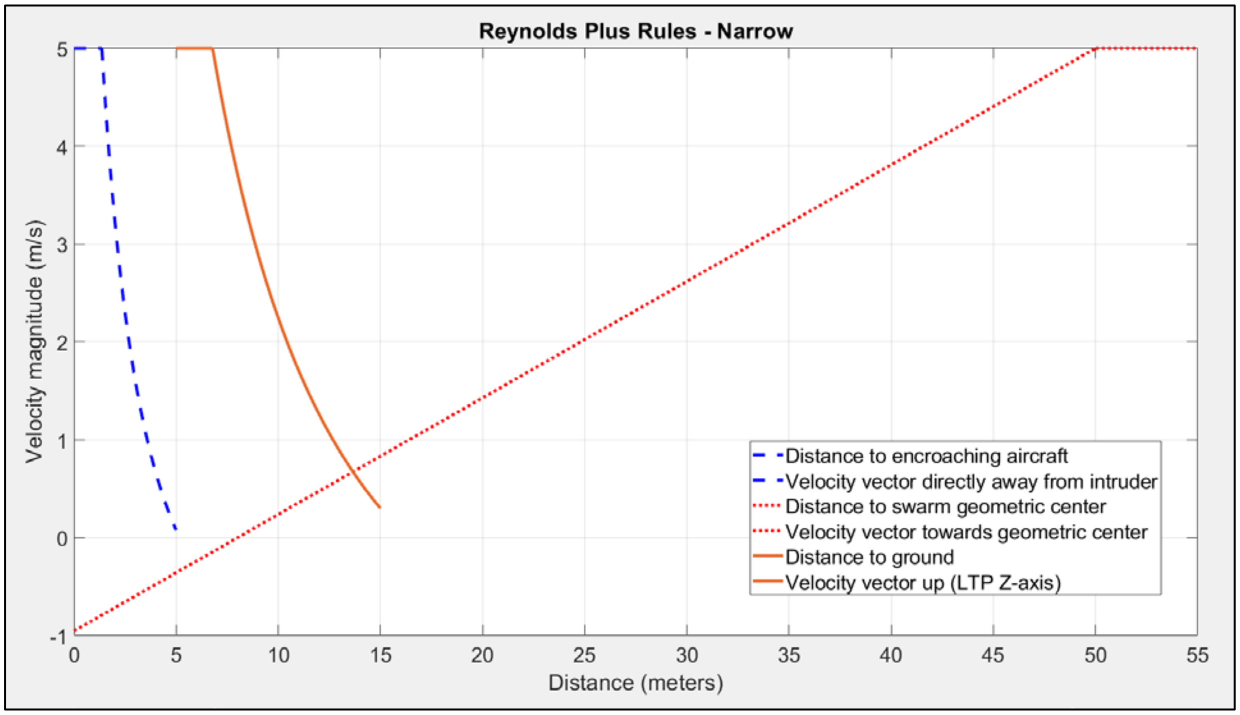


Figure 21. Reynolds+ Rule Magnitude Plots, Narrow Version

Software in the Loop (SITL) Testing

SITL was used extensively to develop the Reynolds+ rules in a low-cost environment not subject to weather or other environmental and logistics constraints. Since the focus of this research is on a testable architecture platform, SITL setup and discussion has been placed in Appendix D.

Metrics

As mentioned in Chapter II, measuring the performance of a swarm of aircraft can be a very subjective task. The primary method would be to evaluate the swarm's performance of some task – surveillance coverage, automated refueling, or target engagement for instance. Since the purpose of this research was to develop a testable architecture baseline, mission evaluation is not feasible. Instead, a proxy measure is used to evaluate safety of flight, and fluctuation of the bucket method implementing Reynolds' rules is explored.

Given the primary risk during flight testing for this research is midair collision between aircraft, frequency plots of the distance between aircraft are generated. The swarm is flown in various patterns first with wider spacing and again with narrow spacing (see Appendix C for governing equations). Distance frequency should peak near the start of the separation radius, drop off sharply as spacing distance decreases, and decrease less sharply as distance increases. This metric may be useful for describing safety of flight in future swarm development as it can shape safety pilot expectations as to how close is "too close," when aircraft are operating in very close proximity.

Results for evaluating the bucket method are less objective, but it is important to see that all the rules are in fact being utilized appropriately, and that no rule is consuming the majority of maneuvering capability when it should not be. Conversely, when a safety rule is under maximum effect, the other rules should be ignored. For example, when aircraft are in close proximity, it should be clear that the separation rule overrides all others – but is quickly resolved so the aircraft can resume mission-related activity.

Test and Verification Procedure

Flight testing will take place at Wright-Patterson Air Force Base, Area B, near Dayton, Ohio. An enclosed cage has been set up near AFIT for initial tests, to ensure the system has basic functionality before open-air tests. The cage, seen in Figure 22, measures 45' x 65' x 40' (LxWxH), is composed of nylon netting with two access panels. A segment of the decommissioned runway is available for open-air testing, pictured in Figure 23.

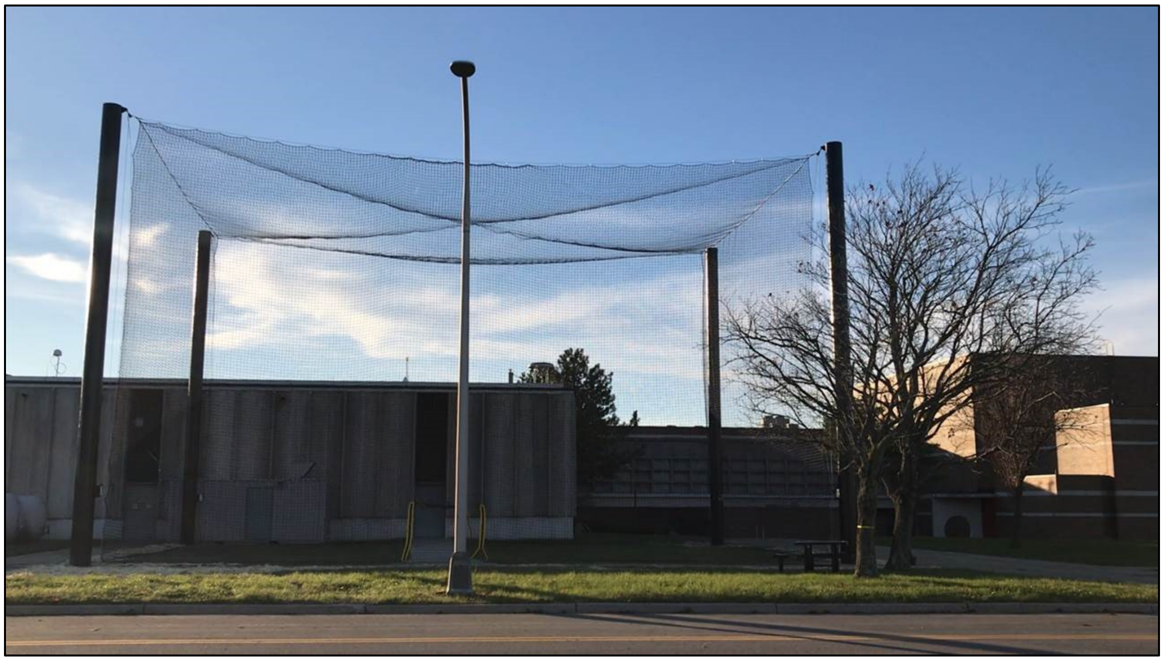


Figure 22. AFIT Small UAV Enclosure

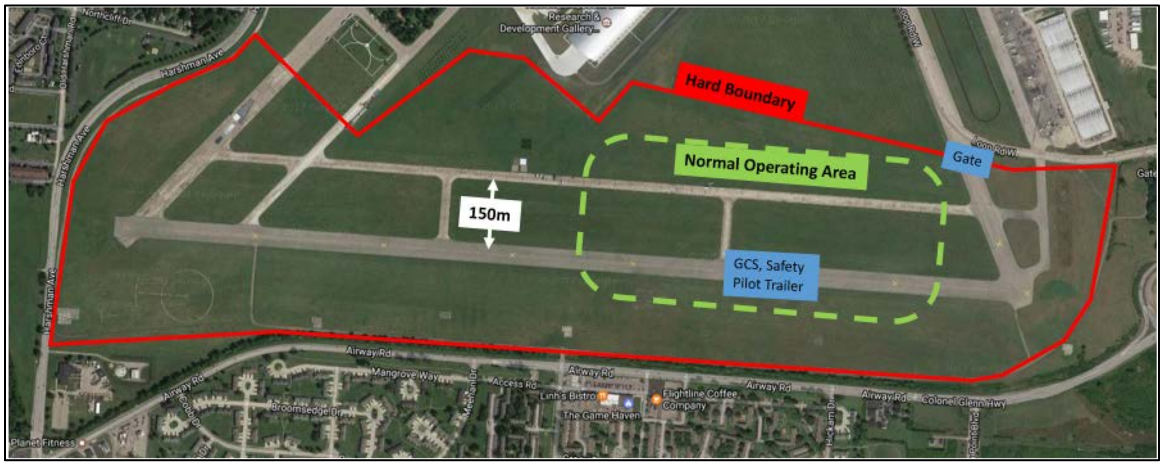


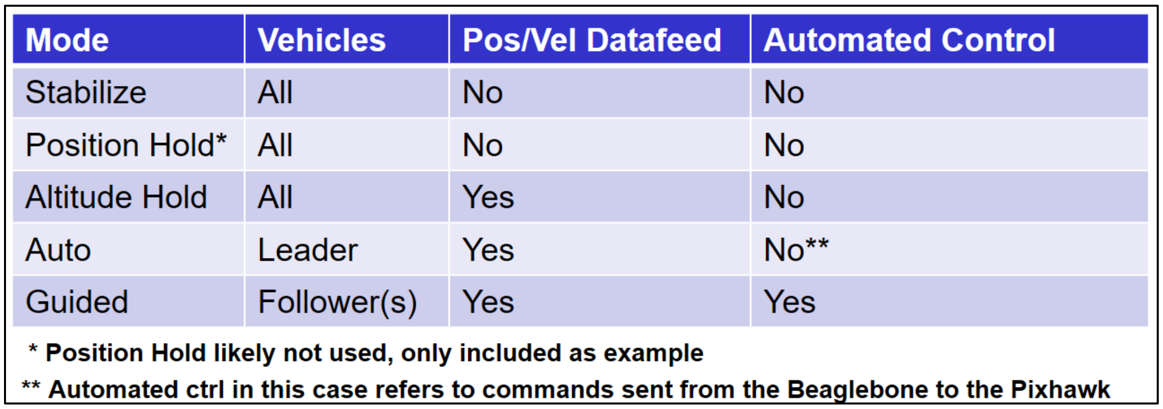
Figure 23. WPAFB Area B Small UAV Flight Operating Area

Although the swarm architecture developed in this thesis only strictly requires four personnel at most to fly three aircraft, eleven will be present to comply with the current flight release during open-air tests. Tests in the cage enclosure do not require the full crew. Four test scenarios have been devised to test the architecture, as well as familiarize ground personnel with swarm operations. These scenarios will be flown first with only two aircraft running the wide spacing described earlier in this document. Then the scenarios will be repeated with three aircraft, also with wide spacing. If the safety pilots are comfortable with the wide spacing and determine it may be safe to proceed with closer spacing, the scenarios will be run twice more with the narrow spacing: once with two aircraft, and once with three. If at any time ground personnel feel the aircraft are liable to collide, the aircraft will be manually recovered immediately, and the safety pilots will determine if the test should be attempted again or not.

Each aircraft will have three personnel responsible for it: a ground control station (GCS) monitoring telemetry from the autopilot, an observer, and a safety pilot. The first two are extraneous for testing purposes but required for safety of flight. The safety pilots play a more active role, described in the next paragraph. In addition to the aircraft crew there is a primary and backup test director. Each scenario will follow a similar script.

The aircraft will be powered on, along with the strap-on guidance packages. Once the guidance packages have connected to the network, the test director will start the Reynolds+ scripting which will enforce swarm behavior in Guided mode. The aircraft will take off under manual control of the safety pilots in Stabilize mode. They will be flown to approximately 20m altitude and spaced 20-30m apart horizontally, and then switched into Altitude Hold mode by the safety pilots. In Altitude Hold, the guidance packages will begin exchanging telemetry with each other but will not be able to maneuver their aircraft. The lead aircraft will remain in Altitude Hold, and the remaining one or two aircraft will be switched into Guided mode by the safety pilots. Once in Guided mode, the follower aircraft will follow the Reynolds+ rules as directed by their on-board script. If at any time one or more aircraft need to be recovered, the responsible safety pilot(s) will place their aircraft in Stabilize mode, which will immediately cut off the telemetry sharing with other aircraft and end any commands issued by the guidance package. Mode behaviors are shown in Table 3. Once each test scenario, or series of scenarios, is complete, the aircraft will be recovered manually.

Table 3. Aircraft Mode Behavior Summary



Mode	Vehicles	Pos/Vel Datafeed	Automated Control
Stabilize	All	No	No
Position Hold*	All	No	No
Altitude Hold	All	Yes	No
Auto	Leader	Yes	No**
Guided	Follower(s)	Yes	Yes

*** Position Hold likely not used, only included as example**
**** Automated ctrl in this case refers to commands sent from the Beaglebone to the Pixhawk**

The “lead” aircraft will be designated ahead of time, but the aircraft will all be running the same script, so it does not strictly matter which particular vehicle it is. The lead aircraft’s safety pilot will have a different configuration on their radio, however. The manual mode switch has three positions. All aircraft will have Stabilize and Altitude Hold modes available. The follower pilots will have Guided on the third position, and the lead pilot will have Auto (waypoint-following).

Cage Testing

Flight tests within the cage enclosure only require a handful of personnel: one safety pilot per aircraft, plus one for ground support. No formal tests were originally planned for this stage; it was intended for troubleshooting and system checkout before open-air tests. However, due to environmental constraints the only useful data was from the final checkout flight within the cage. Enclosed flights involved a series of tests, starting with placing one swarm member on the ground, broadcasting with motors off, turning on an airborne swarm member outside nominal equilibrium distance from the grounded aircraft, and looking for appropriate behavior. Note the aircraft do not have to be airborne to broadcast telemetry, they merely have to be turned on with a good GPS fix in Altitude Hold mode. Another test included "dragging" or "pushing" a swarm member around the cage with a leader aircraft (two vehicles airborne). The last flight included one swarm member on the ground, the lead vehicle manually flown in an upper corner of the cage, and a third aircraft set to guided mode on the opposite side of the enclosure. This list isn't exhaustive; the purpose of cage testing was to provide indications the Reynolds+ algorithm and the architecture as a whole is functioning well enough to execute open-air tests.

Data collection was entirely on-board the aircraft through the Pixhawk telemetry log and LCM's innate data logging. The data had to be manually retrieved and decoded post-flight, although a stationary ad-hoc node on the ground could be used to collect LCM traffic. The X-8 multirotor aircraft in the given configuration were expected to have 15-20 minutes of flight time.

Summary

Although open-air testing was planned, all data was collected in an enclosed environment. One aircraft was designated the “leader” and the remaining two aircraft “followers.” Data collection was conducted on-board: the Pixhawk autopilot stored telemetry data, and the Beaglebone Black companion computer stored all LCM traffic sent and received. Each aircraft retained a record of all LCM traffic plus its own telemetry as recorded by the autopilot. The data will be analyzed in Chapter IV, with particular attention to the separation distance as a frequency plot, and the distribution of each rule within the velocity commands sent to the autopilot from the companion computer on the airborne vehicle in guided mode.

IV. Analysis and Results

Chapter Overview

This chapter discusses the procedures and results of the testing methods described in Chapter III. It evaluates the architecture performance in quantitative and qualitative terms, and provides answers to investigative questions from Chapter I.

Test Scenario

After powering on the autopilots and guidance packages, autonomous scripts were started via a fourth node on the mesh ad-hoc network. The fourth node was connected to each of the vehicles via secure shell (SSH) and was used to launch the autonomy software and data logging in each of the three vehicles. This was done with a screen command to reduce bandwidth between the fourth node and the three vehicles. With the autonomy software started, the SSH connections were closed and control of the vehicles was entirely in the hands of the safety pilots or the software when in guided mode.

One aircraft (Vehicle 3) was placed on the ground, roughly near the center of the enclosure, and set to altitude hold, allowing it to broadcast as a swarm member even though it would remain stationary. The lead aircraft (Vehicle 1) was manually flown to the northeast corner of the enclosure, as close to the roof netting as was deemed reasonably safe given weather conditions and placed in altitude hold – also broadcasting as a swarm member though still under manual control. The last aircraft (Vehicle 2) was manually flown to the southern side of the enclosure, approximately centered between the east and west sides, and halfway between the ground and roof. There it was placed in altitude hold to begin processing data from the other two aircraft, then set to guided mode for autonomous movement. The position data of all vehicles for the flight duration, as recorded by the LCM log on-board Vehicle 2 is shown in Figure 24. The units for Figure 24 are in meters, measured from a predetermined reference point on the WPAFB Area B runway in a local East-North-Up (ENU) frame.

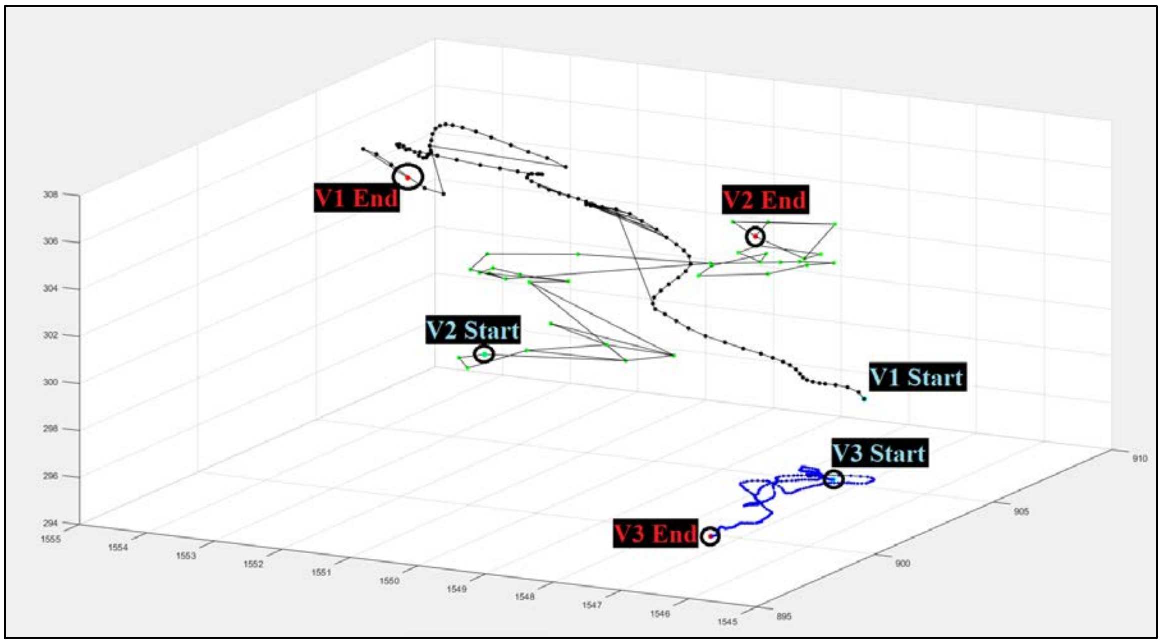


Figure 24. Cage Flight Position Log

Limitations

Vehicle 3 was stationary for the duration of the test, yet its LCM traffic (displayed in Figure 24) shows movement, indicating some GPS errors are present. The cage enclosure is located on top of a hill and immediately adjacent to a building of similar height. Wind gusts are frequent, which produced barometric (and thus altitude) errors. The building itself blocks some GPS signals, and creates multipath issues for others, so the position accuracy is not as good as an open-air test away from structures.

Approximately ten seconds after Vehicle 2 was set to guided mode, a gust of wind caused Vehicle 1 to make an incorrect altitude adjustment and strike the roof of the enclosure. Its erroneous position data was broadcast across the network, causing Vehicle 2 to respond as required by the onboard software, although not in a manner necessarily desirable. The safety pilots recovered both airborne vehicles after 18.84 seconds of guided behavior, and some attempts were made to continue testing but the environmental conditions were deemed unsafe for further flight.

Based on the test performed, the expectation is Vehicle 2 will show it started further from both vehicles, and moved to close the gap, finding a dynamic equilibrium point on a line between both vehicles. Because the lead aircraft is flying and not truly stationary, the distance between Vehicles 2 and 1 should exhibit higher variation than between Vehicles 2 and 3. The distance from Vehicle 2 to Vehicle 1 should also be smaller than the distance to Vehicle 3 because the second Reynolds+ rule introduced (flight deck) should prevent it from moving too close to the ground. If cohesion towards the geometric center of the swarm would move Vehicle 2 too far below the flight deck, then the velocity commands should reflect rule 1 (if applicable) and rule 2 filling the velocity bucket and preventing the remaining rules from influencing the vehicle's motion.

Quantitative Results

The spacing for the duration of the flight from Vehicle 2 (flying autonomously) to the other two vehicles in the swarm is shown in Figure 25 and summarized in Table 4. Start time at 21.81 seconds indicates when Vehicle 2's mode was changed to Altitude Hold; 0.35 seconds later it was changed to Guided and began automated movement. Summary computations only encompass the 39.91 seconds of flight during which Vehicle 2 was flying autonomously, although position data for all vehicles was collected for 58.03 seconds, and the two non-autonomous vehicles for 79.84 seconds (inclusive).

Table 4. Summary Statistics

Vehicle 2 to:		**Distance (meters)**		
	Status	**Mean**	**StdDev**	**Range**
Vehicle 1	Stationary/Flying	8.1199	2.0094	7.3089
Vehicle 3	Stationary/Grounded	8.4525	1.4349	5.5579

The summary statistics show what was expected: the mean distance from the autonomous vehicle to the lead vehicle was smaller than to the stationary vehicle, and the standard deviation higher due to the lead vehicle's movement. Also visible in Figure 25 the two spacing measures begin diverged for the first ~20 seconds of autonomous flight and roughly converge in the last 20 seconds, demonstrating that a dynamic equilibrium has been achieved, as intended and as predicted by SITL testing. It would not be appropriate to compare the spacing distance to the separation radius as originally desired, because the nominal equilibrium point falls outside that distance from either aircraft. Figure 26 was expected to show a cluster of events near the equilibrium distances (9 to 11 meters) but due to the relatively short collection time this phenomenon is not readily observed.

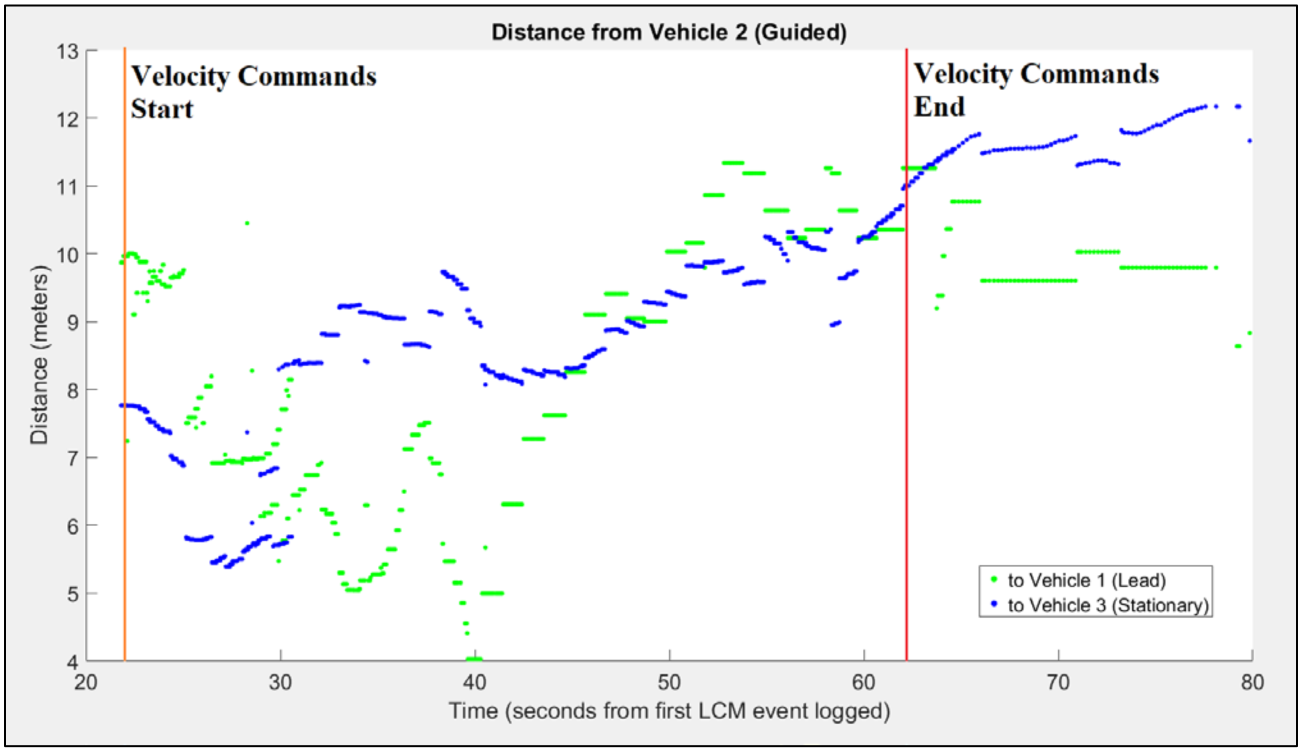


Figure 25. Distance from Vehicle 2 (Guided)

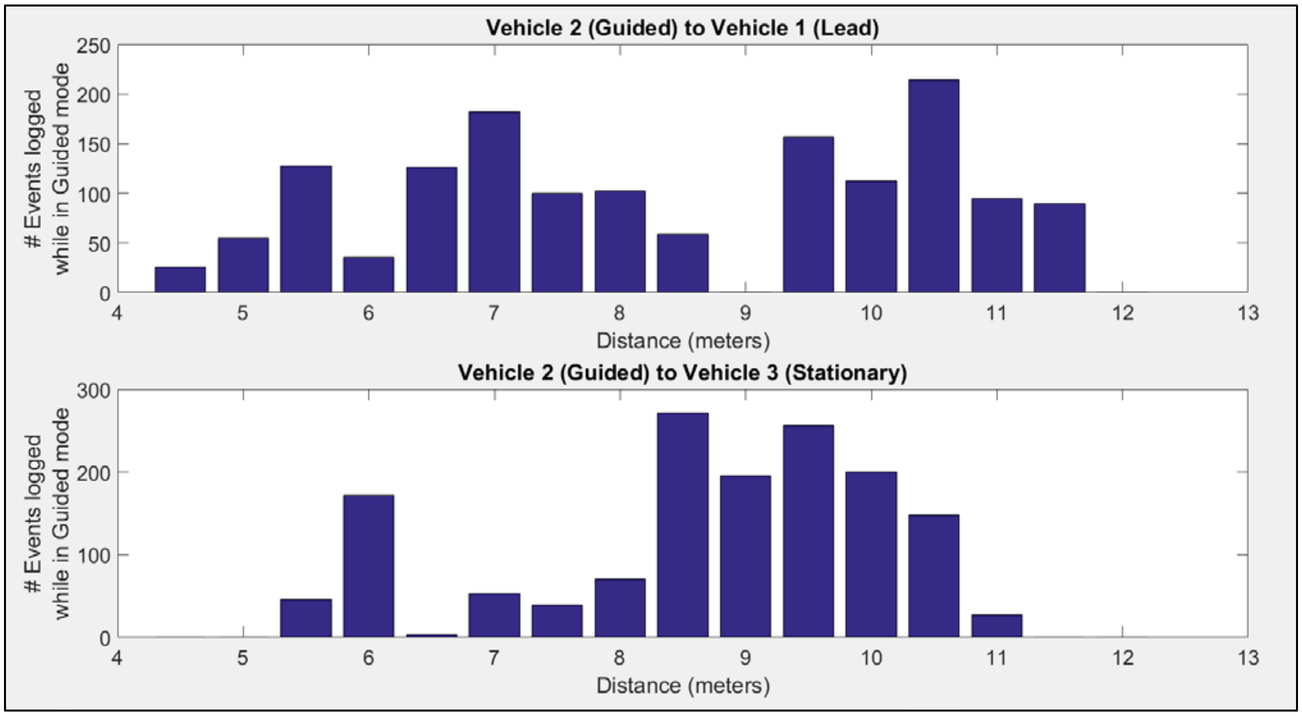


Figure 26. Distance Frequency Plot

The velocity command magnitudes sent to Vehicle 2, broken out by magnitude, are shown in Figure 27. This plot shows the velocity commands are doing exactly what they are supposed to do. The commands do not exceed 2 m/s (reduced from 5 m/s for safety purposes in an enclosed environment), although they can be lower. Higher-priority rules can prevent lower-priority rules from contributing, as seen from ~23 seconds to ~31 seconds where Vehicle 2 drifted below the 4m flight deck and was forced back above by Rule 2. Near 20 seconds into the guided commands (~40 seconds after the first logged LCM message), Figure 25 shows Vehicle 2 moving within 5 meters of Vehicle 1 which should trigger the separation rule (this was also reduced from 10-meter spacing for enclosed flight), and input from Rule 1 is shown in Figure 27 at that time.

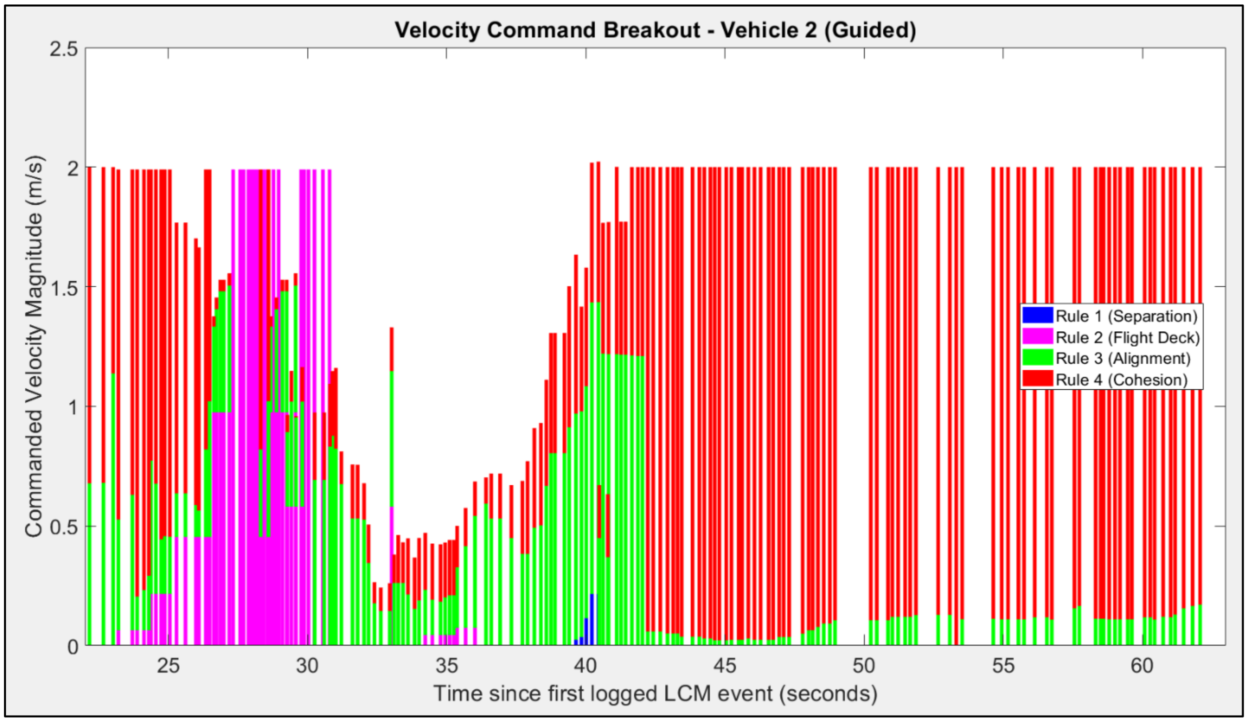


Figure 27. Velocity Command Magnitudes by R+ Rule

Also of note in Figure 27 is the sudden drop in Rule 3 (Alignment) contribution around 42 seconds into collection. This likely indicates when wind gusting and/or GPS errors began to cause Vehicle 1 to broadcast position changes with no accompanying velocity information. This sort of event was not replicated in SITL because the virtual sensors in the software environment are not subject to error – the velocity data would have been broadcast proportional to the change in position. Ultimately, based on the data collected, the automated vehicle responded as intended and as expected.

Qualitative Results

This section will evaluate the tested architecture's performance from a qualitative standpoint. Looking back at the supervisor-subordinate roles, the architecture was as flexible as intended. The safety pilots, although ultimately able to seize control of any of the vehicles at any time, were able to change the roles of the individual aircraft from sub-supervisor to subordinate with the "flick of a switch." The aircraft immediately assumed the assigned roles. The aircraft were all in a subordinate role on takeoff, and both the lead and stationary aircraft served as supervisors to the autonomous aircraft. Once the autonomous aircraft was changed to Guided mode, it became the subordinate of the other two aircraft and its movement subject to their broadcast position and velocity data. The lead aircraft served as a sub-supervisor as it was still subordinate to safety pilot manual control, but otherwise supervised the movement of the autonomous aircraft. The stationary aircraft also served as a sub-supervisor as its movement was restricted to a point on the ground, but in an unenclosed flight test would have served as a peer to the autonomous aircraft since it too would have been flying autonomously. The aircraft were all capable of role changes with a single switch, which was an intended result of this research.

Based on the test performed, the autonomous aircraft was expected to fly towards the geometric center of all three aircraft, which would have shifted proportional to the aircrafts' movement. The autonomous aircraft was expected to settle directly between the lead aircraft's nominal position and the stationary aircraft, with some small oscillations due to broadcast position and sensor errors, and the slight movement of the lead aircraft. The autonomous aircraft did exactly what was expected: it flew directly to a point approximately halfway between the other two aircraft, with some adjustment closer to the lead vehicle so as not to stay below the designated flight deck.

The guidance packages connected to the ad-hoc network seamlessly on startup, with an estimated 1-2-minute delay. Earlier tests resulted in communications back-log and an unresponsive system, so the position/velocity broadcast rate and the velocity command generation rate were reduced from ~20 Hz to approximately 10 Hz. This reduction, combined with starting the software in screens separate from the SSH terminals used to start the software, yielded a smooth system launch with no observed communication delays or dropouts due to flooded channels. The safety pilots observed that the broadcast and command rate could probably be reduced to 1-2Hz with no change in performance, based on their own experience and reaction times. Safety pilot observations of the autonomous behavior itself were that it moved at a rate consistent with their capability to recover from a mishap and did not appear to be in danger of allowing any collision. Wind gusts and altitude errors were of higher concern.

Analysis Summary

Although only one aircraft was flown autonomously, and one aircraft was stationary on the ground rather than flying, all three aircraft simultaneously broadcasted and received their position and velocity data during flight test. This was demonstrated at a transmission rate suitable for flying at velocities up to 2m/s and with a separation radius of 5 meters. Safety pilot observations suggest the transmission rate of position/velocity information could be lowered by one order of magnitude, which would allow for more aircraft to transmit (compared to the original rate) and/or transmission of additional sensor data. The Reynolds+ rules functioned properly, guiding the autonomous aircraft to a position of dynamic equilibrium. The velocity bucket method of rule prioritization also functioned as intended.

Chapter Summary

In this chapter, procedures and results of the test plan described in Chapter II were explored. Changes from the original design were discussed. Chapter V will provide concluding remarks, answers to the investigative questions from Chapter I, and recommendations for future research.

V. Conclusions and Recommendations

Chapter Overview

This chapter presents conclusions about the architecture design and the architecture as tested. Investigative questions from Chapter I are answered, and the significance of this research along with recommendations for future work are described.

Conclusions of Research

The goal of this research was to create a testable architecture for a swarm of multi-rotor aircraft to cooperatively navigate, with or without guidance from outside the swarm. The swarm should consist of at least three aircraft, be scalable to include more, and at a minimum share position and velocity data to enable close-proximity flight up to navigation instrumentation error without collision. The communication segment of the architecture should also be able to accommodate sharing of sensor data in future iterations. Although environmental conditions did not permit a full open-air flight test, such data as could be collected in an enclosed flight suggest this research was successful.

Investigative Questions Answered

What is one architecture that supports collaborative communication between three or more multirotor air vehicles that can be scaled to include more?

The architecture developed by this research can execute autonomous missions through collaborative communication and can be scaled to include more aircraft and sensor payloads. It was successfully tested at a command execution and data transmit/receive rate of 10Hz with three aircraft. The primary components are any multirotor aircraft whose autopilot can receive velocity commands in a local NED (or ENU) frame, a companion computer, and a Wi-Fi adapter set up to connect to a mesh ad-hoc network. The specific missions are not prescribed; its very purpose is to be a baseline from which many kinds of missions can be tested by adding components and software to the architecture.

The architecture can best be described as a layered software pattern with four layers, shown in Figure X. These are: the layer 1-2 networking (in this case the mesh ad-hoc network), the messaging layer which shares telemetry and other data, the autonomy layer governing behavior and storing data, and the vehicle abstraction layer which controls the vehicle hardware. This layered architecture provides flexibility in implementation: each layer is modular and not tied to any specific hardware or software item. The layers work in conjunction to perceive the environment, determine a course of action, and execute the action.

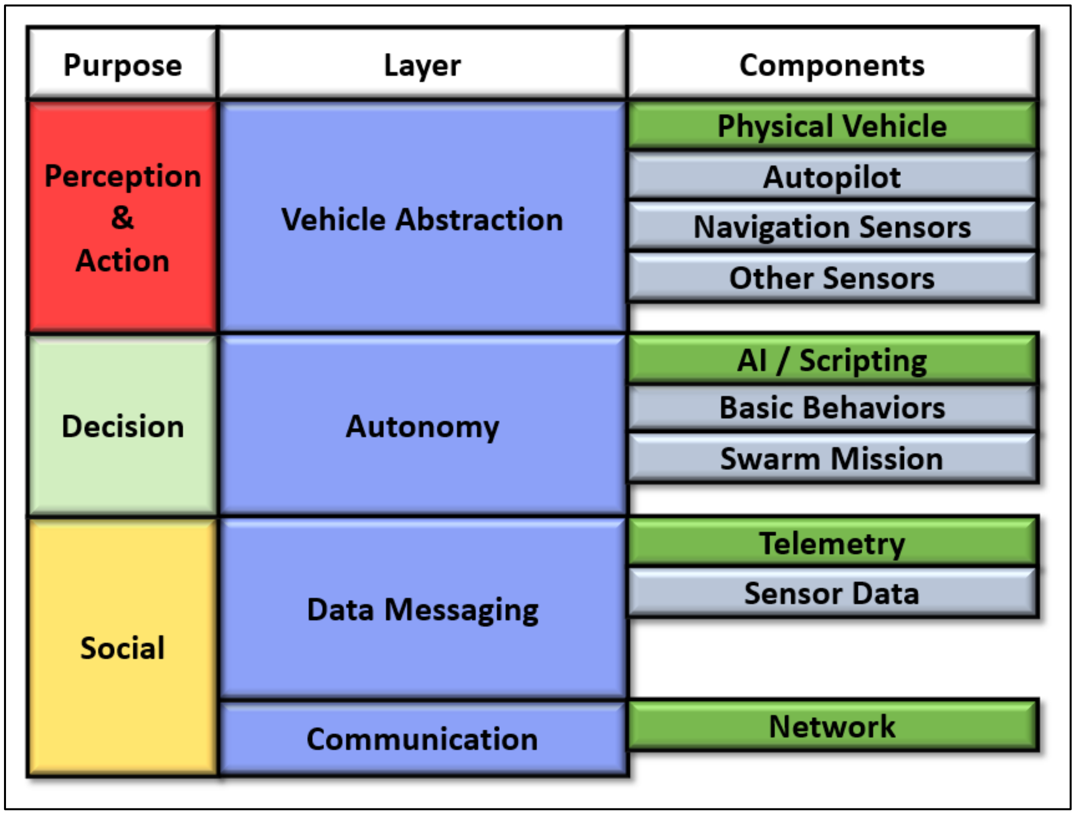


Figure 28. Swarm Architecture

What is the distribution of separation distances between vehicles implementing a version of Reynolds' flocking rules and how does it change with different parameter settings and flight patterns?

This question is partially answered from the cage test of the architecture. The vehicles do establish a dynamic equilibrium when two of the vehicles are stationary or nearly-stationary, where the third vehicle is driven by the governing rules to a stable location between the other two and maintains its position to within the error bounds of its navigational sensors.

What is the contribution of velocity commands by rule using a prioritized velocity bucket instead of a more traditional weighted-rule method?

This architecture's implementation and test of the velocity magnitude bucket frequently led to "overflow" of the bucket, where the desired magnitude was greater than the limit, so the rules were frequently saturated. This is likely due to the low velocity limit of 2m/s enforced in the enclosed cage, and might be different in an open-air test with a higher limit of 5m/s. The lower-priority rules governed aircraft behavior most of the time. The higher-priority rules were enforced when needed, with a swift enough response to permit the lower-priority rules to resume governing autonomous behavior when ground-collision or vehicle-collision avoidance rules were no longer in activation radius.

Significance of Research

This is the first complete architectural description of a multi-rotor aircraft swarm that can be built with COTS components and OSS coding. It's been demonstrated to work by both simulation and real-world flight. It also has room to expand to include more than three aircraft in the swarm and can leverage LCM messaging flexibility to pass data from sensors added to one or more aircraft. The behavior patterns of the aircraft can be modified with a few lines of code to incorporate formation flight, or flight patterns in reaction to sensor data. This will provide time-saving value for future real-world tests of swarming algorithms and utilities.

Recommendations for Action

This architecture should be tested in various flight patterns including: leader-following to ensure movement of the swarm as a single unit, leader-chasing to ensure collision-avoidance for safety-of-flight purposes, and leader-waypoint-following to demonstrate swarm independence of ground control. The software should be improved by modularizing its components, especially the Reynolds+ rule computations, to enable addition of further rules, and/or removal of existing rules as needed.

Recommendations for Future Research

Future research should continue along several fronts. First, the full capabilities of the architecture should be investigated to determine how many aircraft can be part of the swarm before communications become degraded. If sensors are added and that data is shared, it will also consume bandwidth and reduce the maximum number of permissible vehicles. The trade space between command execution rate, data transmission rate, and number of aircraft/sensors should be explored. It may be possible to accomplish this research with partial hardware-in-the-loop (HITL) testing, where ten or twenty guidance packages are each connected to a simulated autopilot rather than real aircraft. This would permit bench-test of the network and messaging limitations.

The specific equations governing the Reynolds+ rules should also be investigated further because they can probably be tuned to provide smoother control, faster convergence, and fewer oscillations. This might also be feasible in SITL because wind gusting and GPS errors can be injected into the simulation, and then those results compared to real-world testing. The relationship between the constants developed in this research for the Reynolds+ rules and settling time / overshoot of swarm equilibrium in various flight modes should be described. Furthermore, an empirical comparison of the weighted-sum versus the velocity magnitude bucket methods should be performed on real hardware.

This swarming architecture only applies to multirotor aircraft due to their ability to rapidly accelerate in any direction, momentum notwithstanding. Some of this research should be applied to fixed-wing aircraft swarms. The velocity magnitude bucket method proved effective and could be integrated into a fixed-wing swarm, where the velocity directions are limited to a cone in front of the aircraft’s body frame, and the size and shape of that cone depending on the aircraft’s maneuvering capabilities.

Lastly, and perhaps most important, the architecture should be used to test swarming algorithms (such as ISR search patterns) that have heretofore only been tested in simulation or have been implemented with only one vehicle. This architecture provides the baseline and can be easily modified for different rules governing autonomous swarming behavior. The Reynolds+ rules provide a starting point for basic maneuvers and safety-of-flight and should be built upon to provide real-world utility.

Bibliography

ArduPilot. (2016). SITL Architecture. Retrieved from http://ardupilot.org/dev/docs/sitl-simulator-software-in-the-loop.html

Bekmezci, I., Sahingoz, O. K., & Temel, Ş. (2013). Flying Ad-Hoc Networks (FANETs): A survey. *Ad Hoc Networks*, *11*(3), 1254–1270. https://doi.org/10.1016/j.adhoc.2012.12.004

Burns, B. S. (2007). Autonomous Unmanned Aerial Vehicle Rendezvous for Automated Aerial Refueling, MS Thesis, Air Force Institute of Technology.

Crouch, T. (2009). *Lighter than air : an illustrated history of balloons and airships*. Johns Hopkins University Press, Baltimore, MD.

Cummings, M., Bruni, S., Mercier, S., & Mitchell, P. (2007). Automation architecture for single operator, multiple UAV command and control. *The International C2 Journal*, *1*, 1–24. https://doi.org/10.1017/CBO9781107415324.004

Diamond, Theodore T., Rutherford, Adam L., Taylor, J. B. (2009). Cooperative Unmanned Aerial Surveillance Control System Architecture, MS Thesis, Air Force Institute of Technology.

DOD. (2014). Command and Control of Joint Air Operations. *US Joint Publication 3-30*.

Drake, S. P. (2002). Converting GPS Coordinates (φλh) to Navigation Coordinates (ENU), Technical Report, DSTO-TN-0432, DSTO Electronics and Surveillance Research Laboratory, Australia.

Duncan, J. (2017). Step by Step. *FAA Safety Briefing*, (June).

Enrica. (2016). Reynolds Flocking Rules. Retrieved from http://enrica.xyz/wp-content/uploads/2016/03/Human-Particle-System-04.png

Fahrney, D. S. (1980, December). The Birth of Guided Missiles. *US Naval Institute Proceedings*, vol. 106, no. 12, pp. 54-58.

Feddema, John T; Robinett, R. D., & Byrne, R. H. (2004). Military Airborne and Maritime Application for Cooperative Behaviors, (September), 1–49.

Federal Aviation Administration; Office of the Secretary of Transportation; Department of Transportation. (2016). Operation and Certification of Small Unmanned Aircraft

Systems. *14 CFR Parts 21, 43, 61, 91, 101, 107, 119, 133, and 183 [Docket No.: FAA-2015-0150; Amdt. Nos. 21-99, 43-48, 61-137, 91-343, 101-9, 107-1, 119-18, 133-15, and 183-16]*, 1–624. Retrieved from https://www.faa.gov/uas/media/RIN_2120-AJ60_Clean_Signed.pdf

Gray, J. (2015). Design and Implementation of a Unified Command and Control Architecture for Multiple Cooperative Unmanned Vehicles Utilizing Commercial Off The Shelf Components, MS Thesis, Air Force Institute of Technology.

Hambling, D. (2016). US Navy Plans to Fly First Drone Swarm This Summer. *DefenseTech*, (January).

Hardy, S. L. (2015). Implementing Cooperative Behavior & Control Using Open Source Technology Across Heterogeneous Vehicles, MS Thesis, Air Force Institute of Technology.

Hauert, S., Leven, S., Varga, M., Ruini, F., Cangelosi, A., Zufferey, J. C., & Floreano, D. (2011). Reynolds flocking in reality with fixed-wing robots: Communication range vs. maximum turning rate. *IEEE International Conference on Intelligent Robots and Systems*, 5015–5020. https://doi.org/10.1109/IROS.2011.6048729

Howard, C. (2013). UAV command , control & communications. *Military & Aerospace Electronics*, *24*(7), 1–22. Retrieved from http://www.militaryaerospace.com/articles/print/volume-24/issue-7/special-report/uav-command-control-communications.html

Huang, A. S., Olson, E., & Moore, D. C. (2010). LCM: Lightweight Communications and Marshalling. *IEEE/RSJ 2010 International Conference on Intelligent Robots and Systems, IROS 2010 - Conference Proceedings*, (Lcm), 4057–4062. https://doi.org/10.1109/IROS.2010.5649358

Justh, EW; Krishnaprasad, P. (2002). A Simple Control Law for UAV Formation Flying. *Engineering*, Maryland Univ College Park Inst for Systems Research, pp. 4-8.

Kaiser, J. N. (2014). Effects of Dynamically Weighting Autonomous Rules in an Unmanned Aircraft System (UAS) Flocking Model, MS Thesis, Air Force Institute of Technology.

Karl, H. (2005). Ad hoc and Sensor Networks Chapter 3 : Network architecture, Computer Networks Group, Universität Paderborn, Germany.

Kumar, P. R. (2002). Ad Hoc Wireless Networks : Analysis , Protocols , Architecture and

Towards Convergence, *1653*, 217–244.

Li, Ming; Harris, John; Chen, Min; Mao, Shiwen; Xiao, Yang; Read, Walter; Prabhakaran, B. (2015). Architecture and protocol design for a pervasive robot swarm communication networks. *Wireless Communications and Mobile Computing*, (February 2015), 421–430. https://doi.org/10.1002/wcm

Li, M., Chen, M., Lu, K., Mao, S., Zhu, H., & Prabhakaran, B. (2008). Robot swarm communication networks: Architectures, protocols, and applications. *3rd International Conference on Communications and Networking in China, ChinaCom 2008*, 162–166. https://doi.org/10.1109/CHINACOM.2008.4684993

Lockheed-Martin. (2017). K-MAX. Retrieved from https://lockheedmartin.com/us/products/kmax.html

Lüssing, L. (2013). batman-adv scalability Layer 2 Mesh Networks - Myths and Risks, (September).

Marsh, N. (2014). Defining the Scope of Autonomy Issues for the Campaign to Stop Killer Robots. *PRIO Policy Brief*, 4.

Mcclanahan, R. L. (2017). Improving Unmanned Aerial Vehicle Formation Flight And Swarm Cohesion By Using Commercial Off The Shelf Sonar Sensors, MS Thesis, Air Force Institute of Technology.

McCollum, B. T. (2017). Analyzing Gps Accuracy Through The Implementation Of Low-Cost Cots Real-Time Kinematic Gps Receivers In Unmanned Aerial Systems, MS Thesis, Air Force Institute of Technology.

McDaid; Oliver; Strong; Israel. (2002). Remote Piloted Aerial Vehicles: An Anthology. *Centre for Telecommunications and Information Engineering, Monash University.*, 1–17. Retrieved from http://www.ctie.monash.edu.au/

Newcome, L. R. (2004). *Unmanned Aviation: A Brief History of Unmanned Aerial Vehicles*. American Institute of Aeronautics and Astronautics, Inc., Reston, VA, p. 133 & 138

Norris, G. (2016). Auto-GCAS Saves Unconscious F-16 Pilot—Declassified USAF Footage. *Aviation Week*, (September), 1–13.

Olfati-Saber, R. (2006). Flocking for multi-agent dynamics systems: Algorithms and theory. *IEEE Trans. Autom. Control*, *51*(3), 401–420.

Portuguese Air Force, (n.d.). Bartolomeu Lourenco de Gusmao, 1–4. Retrieved from http://earlyaviators.com/egusmao.htm

Philips, Adrian N.; Mullins, Barry E.; Raines, Richard A.; Baldwin, R. O. (2009). A secure group communication architecture for autonomous unmanned aerial vehicles. *Security and Communication Networks*, *2*, 55–69.

Pilania, V. K., Mishra, S., Panda, S., & Mishra, A. (2009). A novel approach to swarm bot architecture. *Proceedings - 2009 International Asia Conference on Informatics in Control, Automation, and Robotics, CAR 2009*, 418–422. https://doi.org/10.1109/CAR.2009.50

Pojda, J., Wolff, A., Sbeiti, M., & Wietfeld, C. (2011). Performance analysis of mesh routing protocols for UAV swarming applications. *Proceedings of the International Symposium on Wireless Communication Systems*, 317–321. https://doi.org/10.1109/ISWCS.2011.6125375

Reynolds, C. W. (1987). Flocks, herds and schools: A distributed behavioral model. *ACM SIGGRAPH Computer Graphics*, *21*(4), 25–34. https://doi.org/10.1145/37402.37406

Rogers, S. (AFRL/RY). (2011). Autonomy, Artificial Intelligence and Human Machine Teaming – Frequently Asked Questions, (June), 3–5.

Saska, M., Vonásek, V., Chudoba, J., Thomas, J., Loianno, G., & Kumar, V. (2016). Swarm Distribution and Deployment for Cooperative Surveillance by Micro-Aerial Vehicles. *Journal of Intelligent and Robotic Systems: Theory and Applications*, *84*(1–4), 469–492. https://doi.org/10.1007/s10846-016-0338-z

Scaramuzza, D., Achtelik, M. C., Doitsidis, L., Fraundorfer, F., Kosmatopoulos, E., Martinelli, A., … Weiss, S. (2014). Vision-Controlled Micro Flying Robots. *IEEE Robotics & Automation Magazine*, (September), 26–40. https://doi.org/10.1109/MRA.2014.2322295

Scharre, P. (2015). Between a Roomba and a Terminator: What is Autonomy? - War on the Rocks. *War on the Rocks*. Retrieved from https://warontherocks.com/2015/02/between-a-roomba-and-a-terminator-what-is-autonomy/

Shibing, Z., & Jianmei, D. (2016). Survey of Media Access Control (MAC) and Routing Technologies of WiFi-MESH wireless Network. *International Conference on Computer and Information Technology Application (ICCITA 2016),* (Iccita), 74–79.

Su, H., Wang, X., & Lin, Z. (2007). Flocking of multi-agents with a virtual leader part II: With a virtual leader of varying velocity. *Proceedings of the IEEE Conference on Decision and Control*, *54*(2), 1429–1434. https://doi.org/10.1109/CDC.2007.4434067

Tanner, H. G., Jadbabaie, A., & Pappas, G. J. (2003). Stable flocking of mobile agents. I. Fixed topology. *42nd IEEE International Conference on Decision and Control (IEEE Cat. No.03CH37475)*, *2*(December 2003), 2010–2015. https://doi.org/10.1109/CDC.2003.1272910

Toscano, L. (2017). Effectiveness of Inter-Vehicle Communications and On-Board Processing for Close Unmanned Autonomous Vehicle Flight Formations, MS Thesis, Air Force Institute of Technology.

USAF, “Air Combat Command.” (2015a). MQ-1B Predator. Retrieved December 8, 2017, from http://www.af.mil/About-Us/Fact-Sheets/Display/Article/104469/mq-1b-predator/

USAF, “Air Combat Command.” (2015b). MQ-9 Reaper. Retrieved December 8, 2017, from http://www.af.mil/About-Us/Fact-Sheets/Display/Article/104470/mq-9-reaper/

USJFCOM J9, (2002). *Swarming Entities Roadmap*, resulting from U.S. Joint Forces Command Joint Experimentation (J9) Swarming Conference, November 4-8, 2002.

Vasarhelyi, G., Viragh, C., Somorjai, G., Tarcai, N., Szorenyi, T., Nepusz, T., & Vicsek, T. (2014). Outdoor flocking and formation flight with autonomous aerial robots BT - 2014 IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2014, September 14, 2014 - September 18, 2014. *arXiv Preprint arXiv:1402.3588*, (Iros), 3866–3873. https://doi.org/10.1109/IROS.2014.6943105

Wilson, D. B., Ali, H. G., & Sukkarieh, S. (2015). Guidance and Navigation for UAV Airborne Docking. *Robotics: Science and Systems*, 1–9. https://doi.org/10.13140/RG.2.1.1496.8162

Woolley, B. G., Peterson, G. L., & Kresge, J. T. (2011). Real-time behavior-based robot control. *Autonomous Robots*, *30*(3), 233–242. https://doi.org/10.1007/s10514-010-9215-y

Zafar, W., & Khan, B. M. (2017). A reliable, delay bounded and less complex communication protocol for multicluster FANETs. *Digital Communications and Networks*, *3*(1), 30–38. https://doi.org/10.1016/j.dcan.2016.06.001

Appendix A – Use Cases

Definitions

Safety Pilot: There is one safety pilot responsible for each X-8 multirotor aircraft. The safety pilots are there to ensure safe operation of the vehicles. If necessary, the safety pilot will take manual control of the vehicle by placing it into altitude hold or stabilize, and then maneuver the vehicle to prevent collision or other undesirable behavior. There are two types of safety pilot: the normal safety pilots act as described above. The lead safety pilot has manual control of the lead X-8 multirotor for most test points and can use it to influence the position of the swarm.

Observer: There is one observer responsible for each X-8 multirotor aircraft. If an observer believes an unsafe or undesirable action is going to occur, it is their responsibility to notify the team swiftly and briefly.

X-8 Multirotor: The X-8 multirotor aircraft compose the swarm vehicles for this thesis. There are two or three in the air depending on the test points being flown. One of the vehicles is the nominal leader, broadcasting as part of the swarm but otherwise under manual control and does not behave autonomously. The remaining one or two vehicles broadcast as swarm members and are controlled autonomously by the Reynolds+ algorithm rules when in guided mode.

GPS: The primary navigation system for the X-8 multirotors, using GPS satellites to determine position and timing.

GCS: There is one ground control station for each X-8 multirotor aircraft. The ground control stations' responsibility is to monitor the telemetry feed off the Pixhawk autopilots. The GCS only interacts with the lead aircraft during one test point where the lead vehicle flies a series of waypoints instead of being controlled manually.

<u>Use Case Example 1</u>

Primary Actors

Non-lead X-8 Multirotors, Non-lead Safety Pilots

Secondary Actors

GPS, GCS, Observers, Lead Safety Pilot, Lead X-8

Pre-Conditions

The X-8 multirotors are airborne in altitude hold mode, spaced approximately 30-50 meters apart, at an altitude of 20-30 meters AGL.

Main Flow

6. The non-lead safety pilots change the mode of their vehicles from altitude hold to guided.
7. The non-lead X-8 multirotors fly autonomously towards the geometric center of the swarm.
8. Once the X-8 multirotors are within 20 meters of each other, their motion changes as the alignment vectors cancel out.
9. The vehicles slow as they approach the center and start to fly apart if they are too close to any other vehicles.
10. This behavior is permitted to continue for a minute or two for sufficient data collection.

Alternative Flow

At any time:

a. A safety pilot overrides automated control.

At any time:

a. A non-lead X-8 multirotor drops below 15m altitude.
b. An additional Reynolds+ rule attempts to elevate the vehicle back above 15m AGL.

At any time:

a. A non-lead X-8 multirotor drops below 5m altitude while in guided mode.
b. The vehicle ceases autonomous behavior and acts as if in position hold mode.

Post-Conditions

The X-8 multirotor vehicles achieve a dynamic equilibrium near the lead vehicle, continuously attempting to move towards the geometric center of the swarm, then repelled by the proximity of other vehicles. Data is collected for a minute or two to characterize the behavior.

Use Case Example 2

Primary Actors

X-8 Multirotors, Lead Safety Pilot

Secondary Actors

GPS, GCS, Observers, Normal Safety Pilots

Pre-Conditions

The X-8 multirotor vehicles have achieved a dynamic equilibrium near the lead vehicle, continuously attempting to move towards the geometric center of the swarm, then repelled by the proximity of other vehicles, and data has been collected.

Main Flow

6. The lead safety pilot maneuvers the lead X-8 multirotor away from the other aircraft.
7. The non-lead X-8 multirotor vehicles autonomously maneuver to follow the lead vehicle.
8. The lead safety pilot maneuvers the lead X-8 multirotor in benign patterns – straight lines or gentle arcs.
9. The non-lead X-8 multirotor vehicles maneuver autonomously in response.
10. This maneuvering is done for a minute or two for sufficient data collection.

Alternative Flow

At any time:

b. A safety pilot overrides automated control.

At any time:

a. A non-lead X-8 multirotor drops below 15m altitude.
b. An additional Reynolds+ rule attempts to elevate the vehicle back above 15m AGL.

At any time:

a. A non-lead X-8 multirotor drops below 5m altitude while in guided mode.
b. The vehicle ceases autonomous behavior and acts as if in position hold mode.

Post-Conditions

The non-lead X-8 multirotor vehicles maneuver in response to the manually-controlled lead vehicle, according to the Reynolds+ algorithm rules. Data is collected for a minute or two to characterize the behavior.

Appendix B – Architectural Function Descriptions

<u>**Function**</u>	<u>**Element**</u>	<u>**Input(s)**</u>	<u>**Output(s)**</u>
Receive GPS signals	Pixhawk + GPS Receiver	L1/L2 PRN signals	Pseudorange to SV
Determine position	Pixhawk + GPS Receiver	Pseudorange to 4+ SV's	Lat, Long, Altitude, Time
Send telemetry to guidance pkg	Pixhawk + GPS Receiver	Position, velocity	Position, velocity
Rcv velocity cmds from guidance pkg	Pixhawk + GPS Receiver	NED velocity, duration	Execute command
Send PWM signals to motors	Pixhawk + GPS Receiver	Desired velocity	PWM signal
Send/rcv GCS radio signals	Pixhawk + GPS Receiver	-	Telemetry
Send/rcv safety pilot radio signals	Pixhawk + GPS Receiver	Manual commands, mode changes	
Fly manually	X-8 Multirotor	Manual commands	Fly as commanded
Fly autonomously	X-8 Multirotor	Waypoint list	Fly waypoints
Communicate w/safety pilot	X-8 Multirotor	Manual commands, mode change	-
Communicate w/GCS	X-8 Multirotor	Waypoint list	Telemetry
Communicate w/other guidance pkgs	Strap-on guidance pkg	LTP position, NED velocity	LTP position, NED velocity
Get telemetry from X-8	Strap-on guidance pkg	Global position, NED velocity	-
Send velocity commands to X-8	Strap-on guidance pkg	LTP positions, NED velocities	NED velocity, duration
Monitor autopilot telemetry	GCS	Telemetry	-
Provide navigation signals	GPS	-	L1/L2 PRN signals
Initiate mode change	Safety Pilot	-	Mode change
Fly vehicle manually	Safety Pilot	-	Manual commands
Monitor AV behavior	Safety Pilot	AV behavior	-
Manual recovery	Safety Pilot	-	Manual commands

GPS – Global Positioning System
LTP – Local Tangent Plane
NED – North, East, Down
PRN – Pseudorange
PWM – Pulse Width Modulation
SV – Space Vehicle

Appendix C – Reynolds+ Rules as Implemented

Velocity Bucket Limit

	Cage	Wide	Narrow
	2 m/s	5 m/s	5 m/s

Rule 1: Separation

	Cage	Wide	Narrow
Effect Begins (d)	5m	10m	5m
Maximum Effect (d)	1.04m	2.35m	1.36m
Mag. Equation	$m1 = \frac{10}{(d+1)^2} - 0.4$	$m1 = \frac{100}{d+7} - 5.7$	$m1 = \frac{100}{(d+2.5)^2} - 1.7$
Variable	*d* = distance to the encroaching aircraft		

The unit vector points directly away from the encroaching aircraft. If multiple aircraft are within the separation radius, the resulting velocities are summed.

Rule 2: Minimum Altitude

	Cage	Wide/Narrow
Effect Begins (d)	4m	15m
Maximum Effect (d)	1.96m	6.79m
Mag. Equation	$m2 = \frac{10}{d^2} - 0.6$	$m2 = \frac{1000}{(d+5)^2} - 2.2$
Variable	*d* = altitude above ground level (m)	

The unit vector points directly away from the ground.

Rule 3: Communication Radius (Not Implemented)

Rule 4: Alignment (same for all variations)

Inclusion Radius: 20m

Mag. Equation:	$v4 = \frac{\sum_{k=1}^{numAcft} v_k}{numAcft}$
Variables	*numAcft* = quantity of aircraft within inclusion radius
	k = generic identifier for each aircraft within inclusion radius
	v_k = velocity of *k*th aircraft within inclusion radius

Rule 5: Cohesion

	Cage	**Wide/Narrow**
Mag. Equation	$m5 = \frac{2*d}{2.75} - 0.909$	$m5 = \frac{5*d}{42} - \frac{20}{21}$
Variable	*d* = distance to the geometric center of all aircraft in the swarm	

The unit vector points directly towards the geometric center of all aircraft in the swarm.

Rule 6: Mission (Not Implemented)

Appendix D – Software in the Loop Setup & Discussion

Introduction

The purpose of this section is to discuss the setup of a software-in-the-loop (SITL) environment in which to develop and test swarming algorithms for the architecture. SITL testing is valuable because it can be accomplished by a single person, whereas flight testing requires a crew of multiple people per aircraft. Furthermore, it does not depend on weather, batteries never need be exchanged, and algorithm corrections can be made nearly on-the-fly. There are some drawbacks however: SITL hardware is often more powerful (faster processor, more memory) than the hardware used in flight, making it difficult to ascertain how quickly information should be exchanged between vehicles. Subtle hardware/software integration issues are hidden as well, for instance the original architecture utilized local coordinates as produced by the Pixhawk autopilot. It turns out that local coordinate frame isn't available until the aircraft is armed, and so global coordinates had to be used along with a short Python script to convert them into a local frame. This was never an issue in SITL. Still, the advantages outweigh the disadvantages when it comes to testing the viability of a given algorithm.

Components

SITL components used in this research were entirely virtual except for the laptop they were running on. Time constraints did not permit the addition of hardware components for a partial hardware-in-the-loop (HITL) test. The components are shown as layered in the table below:

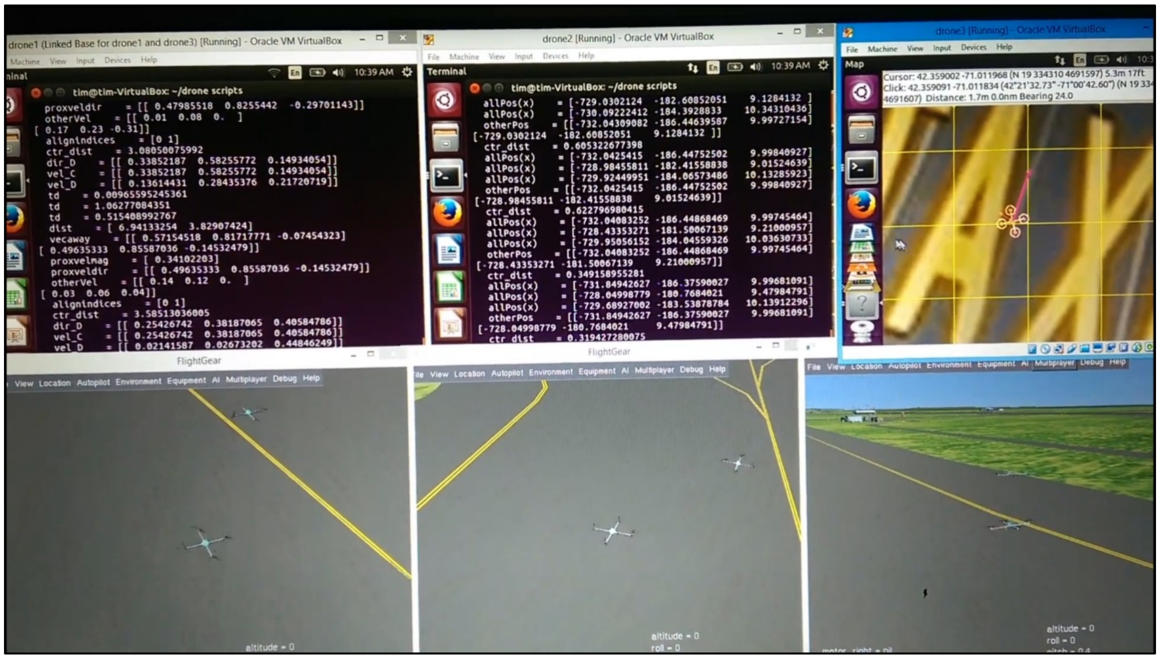
Table 5. SITL Layers

Layer	Component	Subcomponents	Qty @ Layer

1	Asus G750JW Laptop	(Layer 2)	1
2	Oracle VirtualBox	Ubuntu 16.04.3	3
	FlightGear 2017.1.3 (Client)	-	3
	Cygwin	fgms-0-x (FlightGear multiplayer server)	1
3	Ubuntu 16.04.3	DroneKit-Python	1
	fgms-0-x	-	1
4	DroneKit-Python	sim_vehicle.py	1
		new_algorithm.py	1

All SITL testing was performed with a single laptop running Windows 8.1. Within the Operating System (OS), three virtual OS's were utilized. Three VirtualBox clients running Ubuntu stood in for the air vehicles: these terminals contained the flight dynamics model (FDM), the virtual autopilot, and the algorithm controlling the virtual aircraft. Each terminal was assigned a FlightGear client running on the host OS (Windows) to display aircraft activities in a virtual 3D environment. A fourth terminal running Cygwin contained the FlightGear multiplayer server. This server allowed the FlightGear clients to communicate with each other and display all three aircraft within each FlightGear client. A video of this setup can be viewed at the link below, and a still image in Figure 24:

https://www.youtube.com/watch?v=M8U0P3IY0nQ



```
tim@tim-VirtualBox: ~/drone scripts
  proxveldir     = [[ 0.47985518  0.8255442  -0.29701143]]
  otherVel       = [[ 0.01  0.08  0.  ]
[ 0.17  0.23 -0.31]]
  alignindices     = [0 1]
  ctr_dist     = 3.08050075992
  dir_D     = [[ 0.33852187  0.58255772  0.14934054]]
  vel_C     = [[ 0.33852187  0.58255772  0.14934054]]
  vel_D     = [[ 0.13614431  0.28435376  0.21720719]]
  td     = 0.00965595245361
  td     = 1.06277084351
  td     = 0.515408992767
  dist     = [ 6.94133254  3.82907424]
  vecaway     = [[ 0.57154518  0.81717771 -0.07454323]
[ 0.49635333  0.85587036 -0.14532479]]
  proxvelmag     = [ 0.34102203]
  proxveldir     = [[ 0.49635333  0.85587036 -0.14532479]]
  otherVel     = [[ 0.14  0.12  0.  ]
[ 0.03  0.06  0.04]]
  alignindices     = [0 1]
  ctr_dist     = 3.58513036005
  dir_D     = [[ 0.25426742  0.38187065  0.40584786]]
  vel_C     = [[ 0.25426742  0.38187065  0.40584786]]
  vel_D     = [[ 0.02141587  0.02673202  0.44846249]]
```

```
tim@tim-VirtualBox: ~/drone scripts
  allPos(x)     = [-729.0302124  -182.60852051    9.1284132 ]
  allPos(x)     = [-730.09222412 -184.3928833    10.34310436]
  otherPos     = [[-732.04309082 -186.44639587    9.99727154]
[-729.0302124  -182.60852051    9.1284132 ]]
  ctr_dist     = 0.605322677398
  allPos(x)     = [-732.0425415  -186.44752502    9.99840927]
  allPos(x)     = [-728.98455811 -182.41558838    9.01524639]
  allPos(x)     = [-729.92449951 -184.06573486   10.13285923]
  otherPos     = [[-732.0425415  -186.44752502    9.99840927]
[-728.98455811 -182.41558838    9.01524639]]
  ctr_dist     = 0.622796980415
  allPos(x)     = [-732.04083252 -186.44868469    9.99745464]
  allPos(x)     = [-728.43353271 -181.50067139    9.21000957]
  allPos(x)     = [-729.95056152 -184.04559326   10.03630733]
  otherPos     = [[-732.04083252 -186.44868469    9.99745464]
[-728.43353271 -181.50067139    9.21000957]]
  ctr_dist     = 0.349158955281
  allPos(x)     = [-731.84942627 -186.37590027    9.99681091]
  allPos(x)     = [-728.04998779 -180.7684021     9.47984791]
  allPos(x)     = [-729.68927002 -183.53878784   10.13912296]
  otherPos     = [[-731.84942627 -186.37590027    9.99681091]
[-728.04998779 -180.7684021     9.47984791]]
  ctr_dist     = 0.319427280075
```

drone1 (Linked Base for drone1 and drone3) [Running] - Oracle VM VirtualBox
drone2 [Running] - Oracle VM VirtualBox
drone3 [Running] - Oracle VM VirtualBox
Terminal
Map
10:39 AM
FlightGear
View Location Autopilot Environment Equipment AI Multiplayer Debug Help
altitude = 0
roll = 0

Figure 29. SITL Setup

Interfaces

The nominal SITL architecture as it was intended to be set up for one aircraft is shown in Chapter 1, Figure 3. The modified architecture allowing for multiple simulated vehicles and multiple display clients connected through a multiplayer server is displayed below in Figure 25. The various ports listed at each interface are a result of ArduCopter instantiations. When sim_vehicle.py is started in the command line, a numerical argument representing the instance may be included, which then adds ten times the instance number to every port used by the instance. Swarming simulation would not be possible on a single machine without this feature.

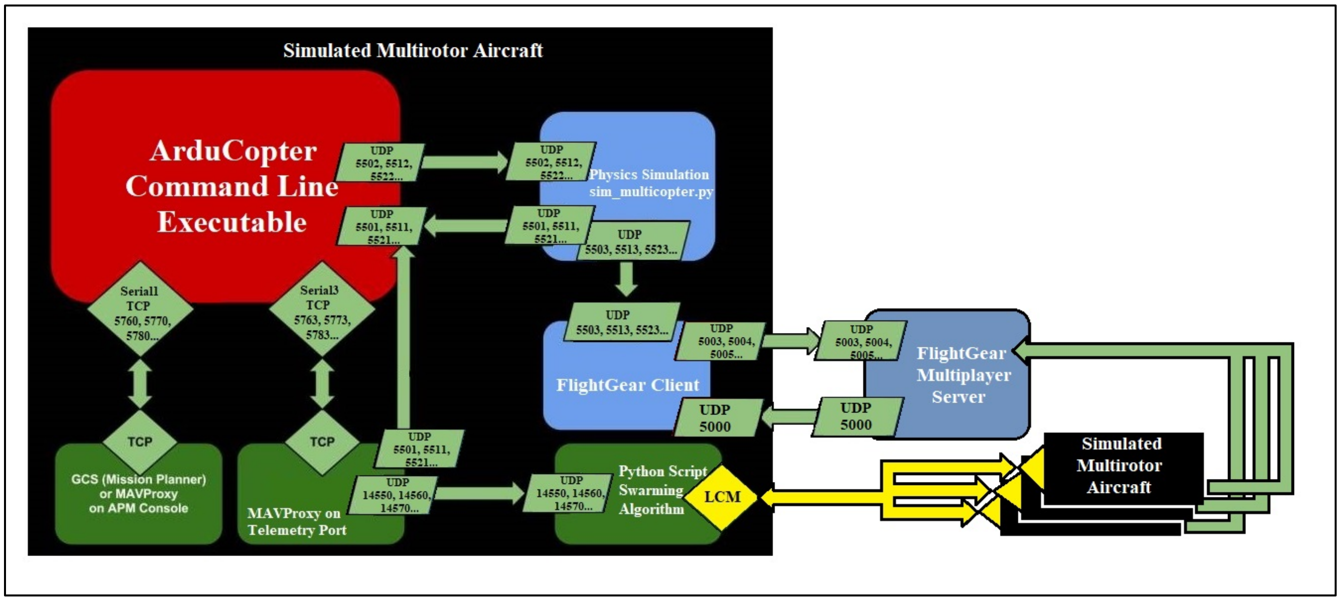

Simulated Multirotor Aircraft
ArduCopter Command Line Executable
UDP 5502, 5512, 5522...
UDP 5502, 5512, 5522...
Physics Simulation sim_multicopter.py
UDP 5501, 5511, 5521...
UDP 5501, 5511, 5521...
UDP 5503, 5513, 5523...
Serial1 TCP 5760, 5770, 5780...
Serial3 TCP 5763, 5773, 5783...
UDP 5503, 5513, 5523...
UDP 5003, 5004, 5005...
UDP 5003, 5004, 5005...
FlightGear Client
FlightGear Multiplayer Server
UDP 5000
UDP 5000
TCP
TCP
UDP 5501, 5511, 5521...
GCS (Mission Planner) or MAVProxy on APM Console
MAVProxy on Telemetry Port
UDP 14550, 14560, 14570...
UDP 14550, 14560, 14570...
Python Script Swarming Algorithm
LCM
Simulated Multirotor Aircraft

Figure 30. SITL Architecture 2.0

Appendix E – Code

This appendix includes all Python and Matlab code written or modified for this research. The Reynolds+ algorithm is shown first, followed by supporting files. Then the analysis tools are provided as well. Note the Reynolds+ algorithm in particular is functional but not optimal – the rules could be modular, and many of the “pre-set” variables could be dynamically updated or input as arguments rather than hard-coded.

Reynolds+ Algorithm (Vehicle 1, Wide Spacing)

rpluswid1.py

```
from dronekit import connect, VehicleMode, LocationGlobalRelative, LocationGlobal
from dronekit import sys, Command
import numpy as np
import lcm
import math
import time
import gpsutils
from datetime import datetime
from pymavlink import mavutil
from exlcm import idposvel
from exlcm import sendvel
import threading
import select
import subprocess
global otherPos
global otherVel
global myPos
global myVel
global homeLoc
global end
global counter
global allPos
global allVel

lc2 = lcm.LCM() #lcm object to handle vehicle2 comms
lc3 = lcm.LCM() #lcm object to handle vehicle3 comms
lcvel = lcm.LCM() #lcm object to record all velocity commands (all vehicles)

end = 0
homeLoc = np.array([39.774185, -84.100031, 0])  # home location must be the same for
all vehicles; this could be dynamic as long as update is time-synchronized
```

```
numAll = 3  # maximum number of vehicles in swarm; preset value
allPos = np.zeros((numAll, 4)) #initialize array of all positions
allVel = np.zeros((numAll, 4)) #initialize array of all velocities

def my_handler(channel, data): #message handler
    global allPos
    global allVel
    msg = idposvel.decode(data) #extract LCM message data
    tempID = int(msg.id) #sender ID
    tempPos = np.array(msg.position) #sender position
    tempVel = np.array(msg.velocity) #sender velocity
    allPos[tempID, [0, 1, 2]] = tempPos #update the all-positions array w/sender data
    allVel[tempID, [0, 1, 2]] = tempVel #update the all-velocities array w/sender data
    ts = time.time() #time of message receipt
    allPos[tempID, 3] = ts #add timestamp to all-positions array
    allVel[tempID, 3] = ts #add timestamp to all-velocities array
    # print("Received message on channel \"%s\"" % channel)
    # print("   id         = %s" % str(msg.id))
    # print("   position    = %s" % str(msg.position))
    # print("   velocity    = %s" % str(msg.velocity))
    # print("")

'''VEHICLE 1 CONNECT INIT'''  # Connect to the vehicle; commented out for bench
testing
print 'Connecting Vehicle 1'  # TCP 232 T-24 IP 192.168.1.11 through 14550
vehicle_connection_string = '/dev/ttyO1' #serial port connection string
vehicle = connect(vehicle_connection_string, wait_ready=False,baud=57600) #connect
print ' '
time.sleep(5)
# Get some vehicle attributes (state) – helps verify connection while troubleshooting
print "Get vehicle #1 attribute values:"
print " GPS: %s" % vehicle.gps_0
print " Battery: %s" % vehicle.battery
print " Last Heartbeat: %s" % vehicle.last_heartbeat
print " Is Armable?: %s" % vehicle.is_armable
print " System status: %s" % vehicle.system_status.state
print " Mode: %s" % vehicle.mode.name  # settable
print " Global Location: %s" % vehicle.location.global_relative_frame

myId = 1 #this changes depending on the vehicle number; should match the ad-hoc IP
myGlobalPos =
np.array([vehicle.location.global_frame.lat,vehicle.location.global_frame.lon,vehicle.loca
tion.global_frame.alt]) #get global location
```

```
myPos =
gpsutils.GeodeticToEnu(myGlobalPos[0],myGlobalPos[1],myGlobalPos[2],homeLoc[0],
homeLoc[1],homeLoc[2]) #convert global location to local frame
myPos = np.array([myPos[1],myPos[0],-1.0*myPos[2]]) #update this vehicle’s position
myVel = np.array(vehicle.velocity) #update this vehicle’s velocity

def send_ned_velocity(velocity_x, velocity_y, velocity_z, duration): #send mavlink
message to the Pixhawk w/commanded velocity
   """
   Move vehicle in direction based on specified velocity vectors.
   """
   msg = vehicle.message_factory.set_position_target_local_ned_encode(
      0,  # time_boot_ms (not used)
      0, 0,  # target system, target component
      mavutil.mavlink.MAV_FRAME_LOCAL_OFFSET_NED,  # frame
      0b0000111111000111,  # type_mask (only speeds enabled)
      0, 0, 0,  # x, y, z positions (not used)
      velocity_x, velocity_y, velocity_z,  # x, y, z velocity in m/s
      0, 0, 0,  # x, y, z acceleration (not supported yet, ignored in GCS_Mavlink)
      0, 0)  # yaw, yaw_rate (not supported yet, ignored in GCS_Mavlink)

   # send command to vehicle once
   #for x in range(0, duration):
   vehicle.send_mavlink(msg) #send the message (velocity command)
   time.sleep(0.1)

def background(): #background: send/receive LCM messages
   global myPos
   global myVel
   global counter
   while counter == 0: #wait until system time is updated
      time.sleep(1)
   timeout = 100  # amount of time to wait, in milliseconds
   tsend = time.time()
   print("tzero: %s" % str(tsend))
   while True: #runs until algorithm is stopped
      vmode = vehicle.mode.name
      print("vmode: %s" % str(vmode))
      if str(vmode) == "GUIDED" or str(vmode) == "ALT_HOLD" or str(vmode) ==
"AUTO": #only broadcast in guided, altitude hold, or auto
         print " Background is updating, broadcasting, and receiving."
         if (time.time() - tsend) > 0.10: #no more than 10hz publish rate
            myGlobalPos = np.array(
            [vehicle.location.global_frame.lat, vehicle.location.global_frame.lon,
             vehicle.location.global_frame.alt]) #get my global position
            myPos = gpsutils.GeodeticToEnu(myGlobalPos[0], myGlobalPos[1],
myGlobalPos[2], homeLoc[0], homeLoc[1],
                                    homeLoc[2]) #convert global to local position
            myPos = np.array([myPos[1], myPos[0], -1.0*myPos[2]])
            myVel = np.array(vehicle.velocity)
            msg = idposvel()
            msg.id = myId
            msg.position = myPos
            msg.velocity = myVel
            lc2.publish("vehicle1", msg.encode()) #send my ID, position, velocity to veh2
            lc3.publish("vehicle1", msg.encode()) #send my ID, position, velocity to veh3
            tsend = time.time() #bookmark send time
         lc2check = lc2.handle_timeout(timeout) check for veh2 messages, handle if avail.
         lc3check = lc3.handle_timeout(timeout) check for veh3 messages, handle if avail.
      else:
         time.sleep(1) #if not in Guided/Alt-Hold/Auto then wait

def foreground(): #this is where the Reynolds+ rules are executed
   global myPos
   global myVel
   global allPos
   global allVel
   global homeLoc
   numAll = 3  # maximum number of vehicles in swarm

   '''DATA FILE INIT'''
   timestr = time.strftime("%m-%d-%Y_%H-%M-%S")  # date-time for file name
   file_name = 'Vehicle_1_' + timestr  # file name appended with date time
   data_file = open(file_name, 'a')  # create txt doc to append to
   print 'Telemetry file open'
   print''

   myGlobalPos = np.array(
      [vehicle.location.global_frame.lat, vehicle.location.global_frame.lon,
vehicle.location.global_frame.alt])
   myPos = gpsutils.GeodeticToEnu(myGlobalPos[0], myGlobalPos[1], myGlobalPos[2],
homeLoc[0], homeLoc[1], homeLoc[2])
   myPos = np.array([myPos[1],myPos[0],-1.0*myPos[2]])
   myVel = np.array(vehicle.velocity)

   allPos = np.zeros((numAll, 4)) #set up position array
   allVel = np.zeros((numAll, 4)) #set up velocity array

   subscription = lc3.subscribe("vehicle3", my_handler) # subscribe to veh3 channel
   subscription2 = lc2.subscribe("vehicle2", my_handler) # subscribe to veh2 channel
```

```
    while True: # loop until script is shut off
        print " Foreground is looping."
        t = time.time() #bookmark time
        otherPos = np.zeros(3)  # set up position array, excluding this vehicle
        otherVel = np.zeros(3)  # set up velocity array, excluding this vehicle
        myId = 1
        myAlt = vehicle.location.global_relative_frame.alt # get height AGL
        print(" myAlt: %s" % str(myAlt))
        allPos[myId, [0, 1, 2]] = myPos #add my most recent position to all-positions
        allVel[myId, [0, 1, 2]] = myVel #add my most recent velocity to all-velocities
        ts = time.time()
        allPos[myId, 3] = ts #timestamp my position
        allVel[myId, 3] = ts #timestamp my velocity

        vel_bucket_max = 2.0 #limit maximum velocity command magnitude
        vel_bucket = vel_bucket_max  # velocity bucket (m/s) reset to max
        alt_limit = 15.0  # flight floor (m above local 0), soft limit
        align_dist = 20.0  # alignment bubble (m) radius

        for x in range(numAll):  # loop through each connection's index
            if sum(allPos[x, [0, 1, 2]]) != 0 and sum(
                    allVel[x, [0, 1, 2]]) != 0 and x != myId:  # check to ensure pos & vel data
isn't empty
                if np.count_nonzero(otherPos) == 0 and np.count_nonzero(otherVel) == 0:
                    otherPos = allPos[x, [0, 1, 2]]
                    otherVel = allVel[x, [0, 1, 2]]
                else:
                    otherPos = np.vstack([otherPos, allPos[
                        x, [0, 1, 2]]])  # append each other vehicle's position parameters to a
central array
                    otherVel = np.vstack([otherVel, allVel[
                        x, [0, 1, 2]]])  # append each other vehicle's velocity parameters to a
central array
        # print("   otherPos    = %s" % str(otherPos))
        # print("   otherVel    = %s" % str(otherVel))
        if np.size(otherPos) == 3:
            numConnections = 1
        else:
            numConnections, num_Cols = otherPos.shape #count current swarm size
        if np.sum(otherPos) != 0 or np.sum(otherVel) != 0:  # wait until at least one other
vehicle's data is received
            allActivePos = np.vstack([otherPos, myPos[0:3]])  # append this vehicle's position
to central array
```

```
        allActiveVel = np.vstack([otherVel, myVel[0:3]])  # append this vehicle's velocity
to central array
        # print("   allActivePos    = %s" % str(allActivePos))
        # print("   allActiveVel    = %s" % str(allActiveVel))
        # implement distance-calculating function to find a 1-D array of distances from
myPos to all other vehicles
        # on the network
        dist = np.array(
            [0])  # initialize empty array - x by 1 array of distances from this vehicle to all
other vehicles
        indices = np.array([0])
        fullMyPos = np.full((numConnections, 3), myPos) # my position, repeated
        dist = np.linalg.norm(fullMyPos - otherPos, axis=1) # distance to each other veh
        #print("   dist    = %s" % str(dist))
        vec_away = (myPos - otherPos) / dist[:,
                                             None]  # x by 3 array of unit vectors from each other vehicle
to this one
        #print("   vecaway    = %s" % str(vec_away))
        vmode = str(vehicle.mode.name)
        if str(vmode) == "GUIDED" and myAlt > 3.0:  #only execute reynolds commands
in GUIDED mode
                                                          # and the copter is already in the air (> 3m alt)

            # implement first Reynolds+ rule: separation
            # initialize desired velocity components
            push_dist = 10.0  # this variable can be tuned; it is this vehicle's separation
bubble radius
            if np.sum(otherPos) != 0:
                indices = np.where(dist <= push_dist)  # get indices of vehicles where dist
<= push distance (10 meters)
                if np.size(indices) != 0:
                    dist_sel = dist[indices]  # select only distance magnitudes from the
indexed list
                    prox_vel_dir = vec_away[indices, :][0]  # select unit vectors from the
indexed list
                else:
                    dist_sel = 100.0
                    prox_vel_dir = np.array([0, 0, 0])
            else:
                dist_sel = 100.0
                prox_vel_dir = np.array([0, 0, 0])
            a = 100  # this variable can be tuned to change the response
            #prox_vel_mag = float(a) / dist_sel ** 2  # magnitude of the response for 5m
bubble
            prox_vel_mag = float(a) / (dist_sel + 7.0) - 5.7
            #print("   proxvelmag    = %s" % str(prox_vel_mag))
            #print("   proxveldir    = %s" % str(prox_vel_dir))
            if np.size(prox_vel_dir) > 3:
                new_vel_A = prox_vel_dir * prox_vel_mag[:, np.newaxis]
            else:
                new_vel_A = prox_vel_dir * prox_vel_mag
            if np.size(new_vel_A) == 3:
                vel_A = new_vel_A
            else:
                vel_A = np.array([sum(new_vel_A[:, 0]), sum(new_vel_A[:, 1]),
sum(new_vel_A[:, 2])])
            mag_A = np.linalg.norm(vel_A)
            if mag_A == 0:
                mag_A = 0.01
            dir_A = vel_A / mag_A
            dir_A = np.squeeze(np.asarray(dir_A))
            if mag_A > vel_bucket:  # limit magnitude of response to bucket size
                mag_A = vel_bucket_max
                vel_A = mag_A * dir_A
                if np.size(vel_A[0]) != 1:
                    vel_A = vel_A[0]
                send_ned_velocity(vel_A[0], vel_A[1], -1 * vel_A[2], 1)
                msg2 = sendvel()
                msg2.velA = (mag_A, dir_A[0], dir_A[1], dir_A[2])
                lcvel.publish("v1vel", msg2.encode()) # broadcast velocity cmd on LCM
                # print("   newVel_A    = %s" % str(vel_A))
            vel_A = mag_A * dir_A
            if np.size(vel_A[0]) != 1:
                vel_A = vel_A[0]
            # end rule 1...
            # implement second Reynolds+ rule: flight floor
            if mag_A < vel_bucket and myAlt < alt_limit:
                vel_bucket = vel_bucket - mag_A  # reduce velocity (magnitude) bucket
                floor_vel_dir = np.array([0, 0, 1])  # vertical unit vector
                #floor_vel_mag = 10 / ((myAlt - 1.4) ** 1.5)  # vertical response magnitude
for 10m floor
                floor_vel_mag = (1000.0 / ((myAlt + 5.0) ** 2.0)) - 0.6 #vertical response
magnitude for 15m floor
                # print("   my alt    = %s" % str(myAlt))
                if floor_vel_mag > vel_bucket:  # limit magnitude of response to bucket size
                    floor_vel_mag = vel_bucket
                    new_vel_B = floor_vel_mag * floor_vel_dir
                    vel_B = vel_A + new_vel_B
                    send_ned_velocity(vel_B[0], vel_B[1], -1 * vel_B[2], 1)
                    msg2 = sendvel()
                    msg2.velA = (mag_A, dir_A[0], dir_A[1], dir_A[2])
                    msg2.velB = (floor_vel_mag, floor_vel_dir[0], floor_vel_dir[1],
floor_vel_dir[2])
                    lcvel.publish("v1vel", msg2.encode())
                    mag_B = vel_bucket_max
                # print("   newVel_B    = %s" % str(vel_B))
                else:
                    new_vel_B = floor_vel_mag * floor_vel_dir
                    dir_B = floor_vel_dir
                    dir_B = np.squeeze(np.asarray(dir_B))
                    vel_B = vel_A + new_vel_B
                    mag_B = np.linalg.norm(vel_B)
                    # end rule 2...
                    # implement third Reynolds+ rule: comm radius
                    # end rule 3...
                    # implement fourth Reynolds+ rule: alignment
            else:
                floor_vel_mag = 0
                floor_vel_dir = np.array([0, 0, 0])
                mag_B = 0
                dir_B = np.array([0, 0, 0])
                dir_B = np.squeeze(np.asarray(dir_B))
                vel_B = vel_A

            mag_C = vel_bucket_max + 1.0 #if rule C runs, this will be replaced.  if not,
rule D will never run
            dir_C = np.array([0,0,0])
            vel_C = mag_C * dir_C
            if mag_B < vel_bucket and floor_vel_mag < vel_bucket:
                vel_bucket = vel_bucket - floor_vel_mag
                align_indices = np.where(dist <= align_dist)[
                    0]  # get indices of vehicles where dist <= alignment bubble radius
                # print("   align indices    = %s" % str(align_indices))
                if np.size(align_indices) == 0:  # if no vehicles within bubble
                    new_vel_C = np.array([0, 0, 0])
                if np.size(align_indices) == 1:  # if one index
                    if np.size(otherVel) == 3:  # if only one other vehicle present
                        new_vel_C = otherVel
                    # print("   newVelC - one veh = %s" % str(new_vel_C))
                    else:  # multiple vehicles, one index within bubble
                        new_vel_C = otherVel[align_indices, :][0]
                        # print("   newVelC - 2+ veh = %s" % str(new_vel_C))
                if np.size(align_indices) > 1:  # multiple vehicles, multiple indices
                    #print("   otherVel    = %s" % str(otherVel))
                    #print("   alignindices    = %s" % str(align_indices))
```

```
                    alignVel = otherVel[align_indices, :]
                    new_vel_C = np.array([np.mean(alignVel[:, 0]), np.mean(alignVel[:, 1]),
np.mean(alignVel[:, 2])])
                mag_C = np.linalg.norm(new_vel_C)
                # print("   mag_C    = %s" % str(mag_C))
                if mag_C == 0:
                    dir_C = np.array([0, 0, 0])
                    dir_C = np.squeeze(np.asarray(dir_C))
                else:
                    dir_C = new_vel_C / mag_C
                    dir_C = np.squeeze(np.asarray(dir_C))
                # print("   dir_C    = %s" % str(dir_C))
                if mag_C > vel_bucket:
                    mag_C = vel_bucket
                    vel_C = mag_C * dir_C + vel_B
                    send_ned_velocity(vel_C[0], vel_C[1], -1 * vel_C[2], 1)
                    msg2 = sendvel()
                    msg2.velA = (mag_A, dir_A[0], dir_A[1], dir_A[2])
                    msg2.velB = (floor_vel_mag, floor_vel_dir[0], floor_vel_dir[1],
floor_vel_dir[2])
                    msg2.velC = (mag_C, dir_C[0], dir_C[1], dir_C[2])
                    lcvel.publish("v1vel", msg2.encode())
                    print(" new_vel_C: %s" % str(vel_C))
                else:
                    vel_C = mag_C * dir_C + vel_B
                    # end rule 4...
                    # implement fifth Reynolds+ rule: flock centering (cohesion)
            # print("   vel_C    = %s" % str(vel_C))
            # print("   mag_C    = %s" % str(mag_C))
            if mag_C < vel_bucket:
                vel_bucket = vel_bucket - mag_C
                if np.size(allActivePos) == 3:
                    flock_center = myPos
                else:
                    flock_center = np.array(
                        [np.mean(allActivePos[:, 0]), np.mean(allActivePos[:, 1]),
np.mean(allActivePos[:, 2])])
                my_ctr_dist = np.linalg.norm(myPos - flock_center)
                #print("   ctr_dist    = %s" % str(my_ctr_dist))
                mag_D = 5.0 * my_ctr_dist / 42.0 - 20.0/21.0
                if my_ctr_dist == 0:
                    dir_D = np.array([0, 0, 0])
                    dir_D = np.squeeze(np.asarray(dir_D))
                else:
                    dir_D = (flock_center - myPos) / my_ctr_dist
                    dir_D = np.squeeze(np.asarray(dir_D))
                if mag_D > vel_bucket:
                    mag_D = vel_bucket
                dir_D[2] = -0.1 * dir_D[2] #dampen vertical cohesion, invert
                vel_D = mag_D * dir_D + vel_C
                #print("   dir_D    = %s" % str(vel_C))
                #print("   vel_C    = %s" % str(vel_C))
                # end rule 5...
                #print("   vel_D    = %s" % str(vel_D))
                if np.size(vel_D[0]) != 1:
                    vel_D = vel_D[0]
                send_ned_velocity(vel_D[0], vel_D[1], vel_D[2], 1)
                msg2 = sendvel()
                msg2.velA = (mag_A, dir_A[0], dir_A[1], dir_A[2])
                msg2.velB = (floor_vel_mag, floor_vel_dir[0], floor_vel_dir[1],
floor_vel_dir[2])
                msg2.velC = (mag_C, dir_C[0], dir_C[1], dir_C[2])
                msg2.velD = (mag_D, dir_D[0], dir_D[1], dir_D[2])
                lcvel.publish("v1vel", msg2.encode())
                print("   newVel_D    = %s" % str(vel_D))
                # implement sixth Reynolds+ rule: swarm direction (move geometric center)
                # end rule 6...

        # send_ned_velocity(velocity_x, velocity_y, velocity_z, duration) # X: North/South,
Y: East/West, Z: Down/Up

        vel_bucket = vel_bucket_max #reset variables for the next loop
        mag_B = vel_bucket_max + 1.0
        mag_C = vel_bucket_max + 1.0
        t2 = time.time()
        for x in range(numAll):
            td = t2 - allPos[x, 3]  # difference between now and timestamp of every position
data point
            #print("   td    = %s" % str(td))
            if td > 2.0:  # if the time difference > 2 sec for a given row
            allPos[x, [0, 1, 2]] = np.array([0, 0, 0])  # zero out the data
            allVel[x, [0, 1, 2]] = np.array([0, 0, 0])  # zero out the data

        time.sleep(0.10 - ((time.time() - t) % 0.10)) #repeat on 10hz interval
        #***End of foreground***

counter = 0

@vehicle.on_message('SYSTEM_TIME') #update the companion computer system time
def listener(self, name, message):
    global counter
    if counter == 0:
        unix_time = (int)(message.time_unix_usec / 1000000)
        dtime = datetime.fromtimestamp(unix_time)
        subprocess.call(["date '%s'" % format(dtime.strftime('%m%d%H%Y.%S'))],
shell=True)
        counter = 1 #update only happens once (instead of ~4Hz)

try:
    b = threading.Thread(name='background', target=background)
    f = threading.Thread(name='foreground', target=foreground)

    b.daemon = True
    f.daemon = True

    b.start() #start background
    time.sleep(3)
    f.start() #start foreground
    while True: time.sleep(100) #allows keyboard interrupt

except (KeyboardInterrupt, SystemExit):
  end = 1
  print '\n! Received keyboard interrupt, quitting threads.\n'
```

gpsutils.py – Global to Local Frame Coordinate Converter

This code was converted from C to Python; original source:

https://gist.github.com/LocalJoost/fdfe2966e5a380957d1c90c462fd1e5c

File location on companion computer: usr/local/lib/python2.7/dist-packages

Code:

```
# Some helpers for converting GPS readings from the WGS84 geodetic system to a local
North-East-Up cartesian axis.

    # The implementation here is according to the paper:
    # "Conversion of Geodetic coordinates to the Local Tangent Plane" Version 2.01.
    # "The basic reference for this paper is J.Farrell & M.Barth 'The Global Positioning
System & Inertial Navigation'"
    # Also helpful is Wikipedia: http:#en.wikipedia.org/wiki/Geodetic_datum

# WGS-84 geodetic constants
import math
import numpy as np
a = 6378137.0;          # WGS-84 Earth semimajor axis (m)
b = 6356752.3142;      # WGS-84 Earth semiminor axis (m)
f = (a - b) / a;          # Ellipsoid Flatness
e_sq = f * (2 - f);    # Square of Eccentricity

# Converts WGS-84 Geodetic point (lat, lon, h) to the
# Earth-Centered Earth-Fixed (ECEF) coordinates (x, y, z).
def GeodeticToEcef(lat, lon, h):
	# Convert to radians in notation consistent with the paper:
	lbda = np.deg2rad(lat)
	phi = np.deg2rad(lon)
	s = math.sin(lbda)
	N = a / ((1 - e_sq * s * s) ** 0.5)
	sin_lambda = math.sin(lbda)
	cos_lambda = math.cos(lbda)
	cos_phi = math.cos(phi)
	sin_phi = math.sin(phi)
	x = (h + N) * cos_lambda * cos_phi
	y = (h + N) * cos_lambda * sin_phi
	z = (h + (1 - e_sq) * N) * sin_lambda
	return np.array([x,y,z])

# Converts the Earth-Centered Earth-Fixed (ECEF) coordinates (x, y, z) to
# East-North-Up coordinates in a Local Tangent Plane that is centered at the
# (WGS-84) Geodetic point (lat0, lon0, h0).
def EcefToEnu(x, y, z, lat0, lon0, h0):
	# Convert to radians in notation consistent with the paper:
	lbda = np.deg2rad(lat0)
	phi = np.deg2rad(lon0)
	s = math.sin(lbda)
	N = a / ((1 - e_sq * s * s) ** 0.5)

	sin_lambda = math.sin(lbda)
	cos_lambda = math.cos(lbda)
	cos_phi = math.cos(phi)
	sin_phi = math.sin(phi)

	x0 = (h0 + N) * cos_lambda * cos_phi
	y0 = (h0 + N) * cos_lambda * sin_phi
	z0 = (h0 + (1 - e_sq) * N) * sin_lambda

	xd = x - x0
	yd = y - y0
	zd = z - z0
	# This is the matrix multiplication
	xEast = -sin_phi * xd + cos_phi * yd
	yNorth = -cos_phi * sin_lambda * xd - sin_lambda * sin_phi * yd + cos_lambda *
zd
	zUp = cos_lambda * cos_phi * xd + cos_lambda * sin_phi * yd + sin_lambda *
zd
	return np.array([xEast,yNorth,zUp])


# Converts the geodetic WGS-84 coordinated (lat, lon, h) to
# East-North-Up coordinates in a Local Tangent Plane that is centered at the
# (WGS-84) Geodetic point (lat0, lon0, h0).
def GeodeticToEnu(lat, lon, h, lat0, lon0, h0):
	ecef = GeodeticToEcef(lat, lon, h)
	enu = EcefToEnu(ecef[0],ecef[1],ecef[2], lat0, lon0, h0)
	return enu
```

idposvel.py – LCM Type Specification for ID, Position, Velocity

```
"""LCM type definitions
This file automatically generated by lcm.
DO NOT MODIFY BY HAND!!!!
"""

try:
   import cStringIO.StringIO as BytesIO
except ImportError:
   from io import BytesIO
import struct

class idposvel(object):
   __slots__ = ["id", "position", "velocity"]

   def __init__(self):
      self.id = 0
      self.position = [ 0.0 for dim0 in range(3) ]
      self.velocity = [ 0.0 for dim0 in range(3) ]

   def encode(self):
      buf = BytesIO()
      buf.write(idposvel._get_packed_fingerprint())
      self._encode_one(buf)
      return buf.getvalue()
```

```
    def _encode_one(self, buf):
        buf.write(struct.pack(">q", self.id))
        buf.write(struct.pack('>3d', *self.position[:3]))
        buf.write(struct.pack('>3d', *self.velocity[:3]))

    def decode(data):
        if hasattr(data, 'read'):
            buf = data
        else:
            buf = BytesIO(data)
        if buf.read(8) != idposvel._get_packed_fingerprint():
            raise ValueError("Decode error")
        return idposvel._decode_one(buf)
    decode = staticmethod(decode)

    def _decode_one(buf):
        self = idposvel()
        self.id = struct.unpack(">q", buf.read(8))[0]
        self.position = struct.unpack('>3d', buf.read(24))
        self.velocity = struct.unpack('>3d', buf.read(24))
        return self
    _decode_one = staticmethod(_decode_one)

    _hash = None
    def _get_hash_recursive(parents):
        if idposvel in parents: return 0
        tmphash = (0x6127d88fd8b7efbf) & 0xffffffffffffffff
        tmphash  = (((tmphash<<1)&0xffffffffffffffff)  + (tmphash>>63)) & 0xffffffffffffffff
        return tmphash
    _get_hash_recursive = staticmethod(_get_hash_recursive)
    _packed_fingerprint = None

    def _get_packed_fingerprint():
        if idposvel._packed_fingerprint is None:
            idposvel._packed_fingerprint = struct.pack(">Q",
idposvel._get_hash_recursive([]))
        return idposvel._packed_fingerprint
    _get_packed_fingerprint = staticmethod(_get_packed_fingerprint)
```

screenLaunch1.sh – Starts Reynolds+ algorithm and LCM log in separate screens

This shell file should be launched on each vehicle via SSH. Two screens are opened, one for the Reynolds+ script, and another for LCM logging. Opening these in new screens prevents the Wi-Fi network from being flooded with unnecessary data, and permits the ground station to SSH into all three vehicles at once with little or no latency.

```
#!/bin/bash

# The goal of this script is to startup each of the individual launch scripts in their own
instance of "screen", detached, so that they will continue to run even after we lose contact
with the plane.

###################### Initial Commands
# Command set 1
# On startup, will need to enter the commands to start LCM:
# sudo ifconfig lo multicast
# sudo route add -net 224.0.0.0 netmask 240.0.0.0 dev lo

# Command set 2
# Need to set the BAUD rate for the ensco radios:
screen -dmS baud /dev/ttyACM1 115200
sleep 5
screen -X -S baud quit

# Command set 3
# Manual screen open and command connection with the Pixhawk (no-GPS), connection
requires sudo for some reason.
# sudo python drivers/px4/px4.py --connect /dev/ttyUSB0 --baud 921600

###################### Open Screens with each script

# May need to start this manually with sudo...
# Python automation script launch file
screen -dmS rplus sh -c "export
LCM_DEFAULT_URL=udpm://239.255.76.67:7667?ttl=2; python rpluscage1.py"

# LCM Logging launch file for the LCM logger function
screen -dmS lcmlog sh -c "export
LCM_DEFAULT_URL=udpm://239.255.76.67:7667?ttl=2; logLaunch.sh"
```

adhoc_startup.sh – Connects to mesh ad-hoc network automatically

This script alone is insufficient to properly set up the ad-hoc network automatically but provides most of the required commands and runs them automatically on startup. Some other files may need configuring depending on the companion computer used.

```
#! /bin/sh
### BEGIN INIT INFO
# Provides:          adhocsetup
# Required-Start:    kmod
# Required-Stop:     kmod
# Default-Start:     2 3 4 5
# Default-Stop:      0 1 6
# Short-Description:
# Description:
### END INIT INFO
# /etc/init.d/adhoc.sh
#

touch /var/lock/adhoc.sh

case "$1" in
  start)
    batctl if add eth0
    ifconfig wlan1 down
    ifconfig wlan1 mtu 1532
    ifconfig wlan1 mode ad-hoc essid reynoldsplus ap CA:FE:C0:DE:F0:0D channel 11
    batctl if add wlan1
    ifconfig wlan1 up
    ifconfig bat0 up
    ifconfig bat0 10.200.8.4
    route add default gw 10.0.99.1
    echo 1 > /proc/sys/net/ipv4/ip_forward
    export LCM_DEFAULT_URL=udpm://239.255.76.67:7667?ttl=1
    stty -F /dev/ttyO1 57600
    ;;
  stop)
    ;;
  *)
    exit 1
    ;;
esac

exit 0
```

logtocsv.py – Converts LCM Log File to Comma Separated Value for Analysis

This code is particular to LCM channels as-named in this research (vehicle1, vehicle2, vehicle3, v1vel, v2vel, v3vel). It requires a log file name as an argument, and outputs one CSV file depending on the input file: either ID/position/velocity data, or velocity command data.

```
import sys
import lcm
import csv
from exlcm import idposvel
from exlcm import sendvel

if len(sys.argv) < 2:
   sys.stderr.write("usage: logtomat <logfile>\n")
   sys.exit(1)

namestr = str(sys.argv[1])  # name of the log file used as argument
file_name1 = 'idposvel_' + namestr + '.csv'  # idposvel csv file name
file_name2 = 'velsend_' + namestr + '.csv'  # vel cmd csv file name

print('starting csv generation')

with open(file_name1, 'wb') as csvfile1:  # create csv

   log = lcm.EventLog(sys.argv[1], "r")
   writer = csv.writer(csvfile1, dialect='excel', delimiter=',')
   writer.writerow(['channel', 'timestamp', 'id', 'position', 'velocity'])

   log.seek(1)
   event1 = log.read_next_event()
   print(' first channel: %s' % str(event1.channel))
   while True:
         try:
                  if event1.channel == "vehicle1" or event1.channel == "vehicle2" or
event1.channel == "vehicle3":
                           msg = idposvel.decode(event1.data)
                        writer.writerow([str(event1.channel), str(event1.timestamp),
str(msg.id), str(msg.position), str(msg.velocity)])
         event1 = log.__next__()
         except StopIteration:
                  print(' reached end of log file')
                  break
   log.close()

print('idposvel csv file created')

with open(file_name2, 'wb') as csvfile2:  # create csv

   log = lcm.EventLog(sys.argv[1], "r")
   writer = csv.writer(csvfile2, dialect='excel', delimiter=',')
   writer.writerow(['channel', 'timestamp', 'velA', 'velB', 'velC', 'velD'])
   log.seek(1)
   event2 = log.read_next_event()
   while True:
          try:
          if str(event2.channel) == "v1vel" or str(event2.channel) == "v2vel" or
str(event2.channel) == "v3vel":
                          msg = sendvel.decode(event2.data)
                          writer.writerow([str(event2.channel), str(event2.timestamp),
str(msg.velA), str(msg.velB), str(msg.velC), str(msg.velD)])
          event2 = log.__next__()
          except StopIteration:
                  print(' reached end of log file')
                  break

   log.close()

print('velocity command csv file created')
print('job complete')r
```

import_velsend.m – Imports velocity command CSV into Matlab

This script will import a CSV containing velocity command data (as output by logtocsv.py above) into Matlab. A similar script for position/velocity data was not functional and that data must be imported manually.

```
%% Import data from text file.
% Script for importing data from the following text file:
%
%    F:\AFIT\Thesis\Code\Analysis\velsend_06-12-2017_16_29_12.00.csv
%
% To extend the code to different selected data or a different text file,
% generate a function instead of a script.

% Auto-generated by MATLAB on 2017/12/18 12:54:23

%% Initialize variables.
filename = 'F:\AFIT\Thesis\Code\Analysis\velsend_06-12-2017_16_29_12.00.csv';
delimiter = ',';
startRow = 2;

%% Read columns of data as strings:
% For more information, see the TEXTSCAN documentation.
formatSpec = '%q%q%q%q%q%q%[^\n\r]';

%% Open the text file.
```

```
fileID = fopen(filename,'r');

%% Read columns of data according to format string.
% This call is based on the structure of the file used to generate this
% code. If an error occurs for a different file, try regenerating the code
% from the Import Tool.
dataArray = textscan(fileID, formatSpec, 'Delimiter', delimiter, 'HeaderLines' ,startRow-
1, 'ReturnOnError', false);

%% Close the text file.
fclose(fileID);

%% Convert the contents of columns containing numeric strings to numbers.
% Replace non-numeric strings with NaN.
raw = repmat({''},length(dataArray{1}),length(dataArray)-1);
for col=1:length(dataArray)-1
   raw(1:length(dataArray{col}),col) = dataArray{col};
end
numericData = NaN(size(dataArray{1},1),size(dataArray,2));

for col=[1,2]
   % Converts strings in the input cell array to numbers. Replaced non-numeric
   % strings with NaN.
   rawData = dataArray{col};
   for row=1:size(rawData, 1);
      % Create a regular expression to detect and remove non-numeric prefixes and
      % suffixes.
      regexstr = '(?<prefix>.*?)(?<numbers>([-]*(\d+[\,]*)+[\.]{0,1}\d*[eEdD]{0,1}[-
+]*\d*[i]{0,1})|([-]*(\d+[\,]*)*[\.]{1,1}\d+[eEdD]{0,1}[-+]*\d*[i]{0,1}))(?<suffix>.*)';
      try
         result = regexp(rawData{row}, regexstr, 'names');
         numbers = result.numbers;

         % Detected commas in non-thousand locations.
         invalidThousandsSeparator = false;
         if any(numbers==',');
            thousandsRegExp = '^\d+?(\,\d{3})*\.{0,1}\d*$';
            if isempty(regexp(numbers, thousandsRegExp, 'once'));
               numbers = NaN;
               invalidThousandsSeparator = true;
            end
         end
         % Convert numeric strings to numbers.
         if ~invalidThousandsSeparator;
            numbers = textscan(strrep(numbers, ',', ''), '%f');
```

```
                numericData(row, col) = numbers{1};
                raw{row, col} = numbers{1};
            end
        catch me
        end
    end
end


%% Split data into numeric and cell columns.
rawNumericColumns = raw(:, [1,2]);
rawCellColumns = raw(:, [3,4,5,6]);


%% Create output variable
velsend1 = raw;
%% Clear temporary variables
clearvars filename delimiter startRow formatSpec fileID dataArray ans raw col
numericData rawData row regexstr result numbers invalidThousandsSeparator
thousandsRegExp me rawNumericColumns rawCellColumns;

%% Begin post-processing
```

processposvel.m – Format ID/Position/Velocity into useful array

```
l = length(idposvel1); %get number of rows
idposvel = zeros(l,8); %create empty matrix
for count = 1:l % one to lower case L
   idposvel(count,1) = idposvel1{count,1};
   idposvel(count,2) = idposvel1{count,2}/1000000 - idposvel1{1,2}/1000000; %convert
time to seconds, first entry is t = 0 seconds
   idposvel(count,3:5) = posvel(idposvel1{count,3});
   idposvel(count,6:8) = posvel(idposvel1{count,4});
end
```

posvel.m – Supporting script for processposvel.m

```
function NED = posvel(str_in)
%POSVEL Summary of this function goes here
%   Detailed explanation goes here
%m0 = strrep(str_in,'"','');
m1 = strrep(str_in,'(','');
m2 = strrep(m1,')','');
m3 = textscan(m2, '%f', 'delimiter',',');
```

```
m4 = m3{1};
NED = [m4(1),m4(2),m4(3)];
end
```

processvelsend.m – Format velocity command into useful array

```
l = length(velsend1); %get number of rows
velsend = zeros(l,18); %create empty matrix
for count = 1:l % one to lower case L
   velsend(count,1) = velsend1{count,1};
   velsend(count,2) = velsend1{count,2}/1000000 - idposvel1{1,2}/1000000; %convert
time to seconds, t = 0 for the position data (this data should start later)
   velsend(count,3:6) = vels(velsend1{count,3});
   velsend(count,7:10) = vels(velsend1{count,4});
   velsend(count,11:14) = vels(velsend1{count,5});
   velsend(count,15:18) = vels(velsend1{count,6});
end
```

vels.m – Supporting script for processvelsend.m

```
function magNED = vels(str_in)
%VELS Summary of this function goes here
%   Detailed explanation goes here
m1 = strrep(str_in,'(','');
m2 = strrep(m1,')','');
m3 = textscan(m2, '%f', 'delimiter',',');
m4 = m3{1};
magNED = [m4(1),m4(2),m4(3),m4(4)];

end
```

modtime.m – Converts LCM timestamp to UTC datetime group

```
function newtime = modtime(t_in) % input time in microseconds since epoch, output
date/time
t = (t_in * 10^(-6))/(3600*24); % convert time since epoch to days
t_ref = datenum('1970','yyyy'); % set epoch reference
t_mat = t + t_ref; % add time of interest to epoch
newtime = datestr(t_mat,'yyyymmdd HH:MM:SS.FFF'); % convert to date time
end
```

plotpos.m – Plots positions from the imported and processed CSV data

```
l = length(idposvel);
s1 = 0;
```

```
s2 = 0;
s3 = 0;
v1 = 0;
v2 = 0;
v3 = 0;
for count = 1:l % one to lower case L
   if idposvel(count,1) == 1
      if s1 == 0
        scatter3(idposvel(count,3),idposvel(count,4),-1*idposvel(count,5),40,'c','filled')
        s1 = 1;
        hold on
      else
         scatter3(idposvel(count,3),idposvel(count,4),-1*idposvel(count,5),20,'k','filled')
      end
      if v1 > 0 && v1 ~= count
         plot3([idposvel(v1,3), idposvel(count,3)],[idposvel(v1,4),idposvel(count,4)],[-
1*idposvel(v1,5),-1*idposvel(count,5)],'k')
      end
      v1 = count;
   end

   if idposvel(count,1) == 2
      if s2 == 0
        scatter3(idposvel(count,3),idposvel(count,4),-1*idposvel(count,5),40,'c','filled')
        s2 = 1;
        hold on
      else
        scatter3(idposvel(count,3),idposvel(count,4),-1*idposvel(count,5),20,'g','filled')
      end
      if v2 > 0 && v2 ~= count
         plot3([idposvel(v2,3), idposvel(count,3)],[idposvel(v2,4),idposvel(count,4)],[-
1*idposvel(v2,5),-1*idposvel(count,5)],'k')
      end
      v2 = count;
   end

   if idposvel(count,1) == 3
      if s3 == 0
        scatter3(idposvel(count,3),idposvel(count,4),-1*idposvel(count,5),40,'c','filled')
        s3 = 1;
        hold on
      else
        scatter3(idposvel(count,3),idposvel(count,4),-1*idposvel(count,5),10,'b','filled')
      end
      if v3 > 0 && v3 ~= count
```

```
            plot3([idposvel(v3,3), idposvel(count,3)],[idposvel(v3,4),idposvel(count,4)],[-
1*idposvel(v3,5),-1*idposvel(count,5)],'k')
        end
        v3 = count;
    end
end
scatter3(idposvel(v1,3),idposvel(v1,4),-1*idposvel(v1,5),30,'r','filled')
scatter3(idposvel(v2,3),idposvel(v2,4),-1*idposvel(v2,5),30,'r','filled')
scatter3(idposvel(v3,3),idposvel(v3,4),-1*idposvel(v3,5),30,'r','filled')
```

plotvel – Plots velocity arrows from the imported and processed CSV data

```
l = length(idposvel);
for count = 1:l % one to lower case L
    if idposvel(count,1) == 1
        quiver3(idposvel(count,3),idposvel(count,4),-
1*idposvel(count,5),idposvel(count,6),idposvel(count,7),idposvel(count,8),'k')
        hold on
    end

    if idposvel(count,1) == 2
        quiver3(idposvel(count,3),idposvel(count,4),-
1*idposvel(count,5),idposvel(count,6),idposvel(count,7),idposvel(count,8),'g')
        hold on
    end

    if idposvel(count,1) == 3
        quiver3(idposvel(count,3),idposvel(count,4),-
1*idposvel(count,5),idposvel(count,6),idposvel(count,7),idposvel(count,8),'b')
        hold on
    end
end
```

plotdist.m – Plots distances between each vehicle

The plot is a frequency chart, showing how many LCM messages were sent within each 0.5 meter increment from zero to 25 meters separation.

```
l = length(idposvel);
v1pos = [0, 0, 0, 0];
v2pos = [0, 0, 0, 0];
v3pos = [0, 0, 0, 0];
dist1 = [0 0];
dist3 = [0 0];
count2 = 1;
```

```
for count = 1:l % one to lower case L
   if idposvel(count,1) == 1
      v1pos = [idposvel(count,3),idposvel(count,4),idposvel(count,5),idposvel(count,2)];
   end
   if idposvel(count,1) == 2
      v2pos = [idposvel(count,3),idposvel(count,4),idposvel(count,5),idposvel(count,2)];
   end
   if idposvel(count,1) == 3
      v3pos = [idposvel(count,3),idposvel(count,4),idposvel(count,5),idposvel(count,2)];
   end
   if sum(abs(v1pos(1:3))) > 0 && sum(abs(v2pos(1:3))) > 0 && sum(abs(v3pos(1:3))) >
0
     % distance of interest in this case is vehicle 2 to 1 and 3 respectively
     dist1(count2,1) = idposvel(count,2);
     dist3(count2,1) = idposvel(count,2);
     dist1(count2,2) = sqrt((v2pos(1)-v1pos(1))^2 + (v2pos(2)-v1pos(2))^2 + (v2pos(3)-
v1pos(3))^2);
     dist3(count2,2) = sqrt((v2pos(1)-v3pos(1))^2 + (v2pos(2)-v3pos(2))^2 + (v2pos(3)-
v3pos(3))^2);
     count2 = count2+1;
   end

   dtv1 = idposvel(count,2) - v1pos(4);
   dtv2 = idposvel(count,2) - v2pos(4);
   dtv3 = idposvel(count,2) - v3pos(4);

   if dtv1 > 1000000
      v1pos(1:3) = [0 0 0];
   end

   if dtv2 > 1000000
      v2pos(1:3) = [0 0 0];
   end

   if dtv3 > 1000000
      v3pos(1:3) = [0 0 0];
   end

end
figure
scatter(dist1(:,1),dist1(:,2),18,'g','filled')
hold on
scatter(dist3(:,1),dist3(:,2),18,'b','filled')
edges = linspace(4,12,17);
edges2 = edges(2:17);
```

```
N1 = histcounts(dist1(:,2),edges);
N3 = histcounts(dist3(:,2),edges);
figure
subplot(2,1,1)
bar(edges2,N1)
subplot(2,1,2)
bar(edges2,N3)
```

histplot2.m – Repeats the histogram from plotdist

The dist1/dist2/dist3 variables should be copied and edited to reflect only the period while velocity commands are being sent. I did this manually and named the respective variables dist1a and dist3a. The specific boundaries should be edited to include the maximum and minimum spacing distances between vehicles.

```
edges = linspace(4,12,17);
edges2 = edges(2:17);
N1 = histcounts(dist1a(:,2),edges);
N3 = histcounts(dist3a(:,2),edges);
figure
subplot(2,1,1)
bar(edges2,N1)
subplot(2,1,2)
bar(edges2,N3)
```

plotbarvel.m – Plots velocity commands over time

The plot shows every velocity command sent, broken out by rule.

```
l = length(velsend);
v1vel = [0, 0, 0, 0, 0];
v2vel = [0, 0, 0, 0, 0];
v3vel = [0, 0, 0, 0, 0];
count1 = 1;
count2 = 1;
count3 = 1;
figure
for count = 1:l % one to lower case L

   if velsend(count,1) == 1
      v1vel(count1,1) = velsend(count,2);
      v1vel(count1,2:5) =
[abs(velsend(count,3)),abs(velsend(count,7)),abs(velsend(count,11)),abs(velsend(count,1
5))];
      if v1vel(count1,2) == 0.01
```

```
            v1vel(count1,2) = 0;
        end
        count1 = count1+1;
    end

    if velsend(count,1) == 2
        v2vel(count2,1) = velsend(count,2);
        v2vel(count2,2:5) =
[abs(velsend(count,3)),abs(velsend(count,7)),abs(velsend(count,11)),abs(velsend(count,1
5))];
        if v2vel(count2,2) == 0.01
            v2vel(count2,2) = 0;
        end
        count2 = count2+1;
    end

    if velsend(count,1) == 3
        v3vel(count3,1) = velsend(count,2);
        v3vel(count3,2:5) =
[abs(velsend(count,3)),abs(velsend(count,7)),abs(velsend(count,11)),abs(velsend(count,1
5))];
        if v3vel(count3,2) == 0.01
            v3vel(count3,2) = 0;
        end
        count3 = count3+1;
    end
end
if sum(abs(v1vel)) > 0
    subplot(3,1,1)
    v1bar = bar(v1vel(:,1),v1vel(:,2:5),1,'stacked')
    set(v1bar,{'FaceColor'},{'b';'m';'g';'r'})
    L1=legend(v1bar, {'Separation','Flight Deck','Alignment','Cohesion'},
'Location','Best','FontSize',8)
end

if sum(abs(v2vel)) > 0
    subplot(3,1,2)
    v2bar = bar(v2vel(:,1),v2vel(:,2:5),1,'stacked')
    set(v2bar,{'FaceColor'},{'b';'m';'g';'r'})
    L2=legend(v2bar, {'Separation','Flight Deck','Alignment','Cohesion'},
'Location','Best','FontSize',8)
end

if sum(abs(v3vel)) > 0
    subplot(3,1,3)
    v3bar(v3vel(:,1),v3vel(:,2:5),1,'stacked')
    set(v3bar,{'FaceColor'},{'b';'m';'g';'r'})
    L3=legend(v3bar, {'Separation','Flight Deck','Alignment','Cohesion'},
'Location','Best','FontSize',8)
end
```

test.bat – Launches 3 FlightGear clients in Windows, connected to MP server

```
set AUTOTESTDIR="C:\cygwin\home\username\ardupilot\Tools\autotest\aircraft"
c:
FOR /F "delims=" %%D in ('dir /b "\Program Files"\FlightGear*') DO set FGDIR=%%D
echo "Using FlightGear %FGDIR%"
cd "\Program Files\%FGDIR%\bin"

start fgfs ^
    --native-fdm=socket,in,10,,5503,udp ^
    --fdm=external ^
    --aircraft=arducopter ^
    --fg-aircraft=%AUTOTESTDIR% ^
    --airport=KBOS ^
    --geometry=650x550 ^
    --bpp=32 ^
    --disable-anti-alias-hud ^
    --disable-hud-3d ^
    --disable-horizon-effect ^
    --timeofday=noon ^
    --disable-sound ^
    --disable-fullscreen ^
    --disable-random-objects ^
    --fog-disable ^
    --disable-specular-highlight ^
    --disable-anti-alias-hud ^
    --wind=0@0 ^
    --multiplay=in,10,127.0.0.1,5003^
    --multiplay=out,10,127.0.0.1,5000^
    --callsign=AFIT-03

timeout /t 30

start fgfs ^
    --native-fdm=socket,in,10,,5513,udp ^
    --fdm=null ^
    --aircraft=arducopter ^
    --fg-aircraft=%AUTOTESTDIR% ^
    --airport=KBOS ^
    --geometry=650x550 ^
    --bpp=32 ^
    --disable-anti-alias-hud ^
    --disable-hud-3d ^
    --disable-horizon-effect ^
    --timeofday=noon ^
    --disable-sound ^
    --disable-fullscreen ^
    --disable-random-objects ^
    --fog-disable ^
    --disable-specular-highlight ^
    --disable-anti-alias-hud ^
    --wind=0@0 ^
    --multiplay=in,10,127.0.0.1,5004^
    --multiplay=out,10,127.0.0.1,5000^
    --callsign=AFIT-02

timeout /t 25

start fgfs ^
    --native-fdm=socket,in,10,,5523,udp ^
    --fdm=null ^
    --aircraft=arducopter ^
    --fg-aircraft=%AUTOTESTDIR% ^
    --airport=KBOS ^
    --geometry=650x550 ^
    --bpp=32 ^
    --disable-anti-alias-hud ^
    --disable-hud-3d ^
    --disable-horizon-effect ^
    --timeofday=noon ^
    --disable-sound ^
    --disable-fullscreen ^
    --disable-random-objects ^
    --fog-disable ^
    --disable-specular-highlight ^
    --disable-anti-alias-hud ^
    --wind=0@0 ^
    --multiplay=in,10,127.0.0.1,5001^
    --multiplay=out,10,127.0.0.1,5000^
    --callsign=AFIT-01
```

Appendix F – Miscellaneous

This appendix includes information critical to repeating this research but is too short and detailed for inclusion in the main body of research.

Pixhawk-Beaglebone Serial Connection

The Telem2 port on the Pixhawk does not have the same input/output as the Telem1 port by default. To enable it as a telemetry port, add an empty file called “uartD.en” to the APM directory within the Pixhawk’s microSD memory card. This is a poorly-documented procedure found in ArduPilot documentation. On the Beaglebone Black, open the capemgr file (/etc/default/capemgr) and change the line “#CAPE” to “CAPE=BB-UART1,BB-UART2” and reboot the Beaglebone. This will enable the Beaglebone’s serial ports because they are not active by default. These two fixes will allow the Beaglebone to receive telemetry off the Telem2 port, send velocity commands to the Pixhawk. Ensure the SERIAL2_BAUD parameter on the aircraft is set to 57; other parameters starting with “SR2_” may need to be tweaked as well.

These fixes also permit the companion computer to connect to the Pixhawk at the same time as a GCS using Mission Planner, which had not been resolved in previous research. Note the python script running the Reynolds+ algorithm running on the companion computer continuously throws an error code ("Exception in message handler for HEARTBEAT; mode 0 not available on mavlink definition") while the GCS is connected at the same time, but both still provide required functionality.

Companion Computer Modules

The following modules need to be added or updated to the companion computer with apt-get or apt-get install:

update, git, libncurses5-ddev, libncursesw5-dev, gawk, subversion, libapache2-svn, openjdk-6-jdk, python-dev, unzip, python-setuptools, python-opencv, python-wxgtk2.8, python-pip, python-matplotlib, python-pygame, python-lxml, software-properties-common, python-software-properties, libxml2-dev, libxslt-dev, firmware-atheros (this depends on what firmware your Wi-Fi adapter requires), batctl, bridge-utils

The following modules need to be added to Python on the companion computer (pip install): lxml, dronekit, numpy (ensure latest version), future, mavproxy, dronekit=sitl, droneapi, pymavlink (version 2.2.6 preferred – uninstall all copies of pymavlink then reinstall the specific version)

When installing lxml and pymavlink, the memory on the Beaglebone was insufficient so a swap file was created to permit installation:

```
dd if=/dev/zero of=/swapfile1 bs=1024 count=524288
mkswap /swapfile1
chown root:root /swapfile1
chmod 0600 /swapfile1
swapon /swapfile1
```

It was removed after installation using:

```
  swapoff -v /swapfile1
  rm /swapfile1
```

Installation of lxml can take a while on the Beaglebone (estimated at one hour).

Timing

Timing is important to the Reynolds+ code because the data broadcast by other aircraft is stored with a timestamp. If data from any aircraft is more than 2 seconds old, it is discarded and zeroed out, which the algorithm treats as a non-broadcasting aircraft.

The Beaglebone Black companion computers used in this research do not have a backup battery to maintain a clock, and thus always boot up at a hard-coded system time.

The Reynolds+ code includes a snippet that obtains GPS time from the Pixhawk and sets the companion computer's system time to match it upon starting up the script. Initial configurations updated the clock every time the Pixhawk updated its clock (~4 Hz), but this proved to cause problems with timestamps, so clock drift is assumed to be acceptable for the duration of flight (~20 minutes at most) and the update is only performed once.

LCM

For LCM to work properly over the ad-hoc network, a specific common URL must be exported using the following command: "export LCM_DEFAULT_URL=udpm://239.255.76.67:7667?ttl=1". Initially this command had to be manually input every time a terminal was opened, but it was later automated with the ad-hoc network setup.

For LCM to work properly between VirtualBoxes, the export command is: "export LCM_DEFAULT_URL=udpm://224.3.29.71:5005?ttl=2". Additionally, the following command must be run on opening each terminal: "route add 224.3.29.71 dev enp0s8" (check ifconfig to see which interface is appropriate – it may be enp0s3 or similar).

Neither the software nor hardware setups would send and receive LCM messages on the same channel at the same time. Therefore, each vehicle's Reynolds+ algorithm creates a LCM object for each other vehicle in the swarm, with a channel dedicated to that vehicle. Each vehicle listens on its own channel and broadcasts on all other channels. To facilitate message handling simultaneously with the swarm algorithm, Python's threading module was utilized to place message handling in the background and swarm execution in the foreground.

FlightGear Multiplayer Server (FGMS) Setup

This segment includes tips for setting up FGMS in Cygwin. The following sites provide instructions for setting up a multiplayer server:

http://wiki.flightgear.org/Howto:Set_up_a_multiplayer_server

http://fgms.freeflightsim.org/README_cmake.html

Pthreads (see setup instructions) were obtained from:

ftp://sourceware.org/pub/pthreads-win32/ (pthreads-w32-2-9-1-release.zip used for this research)

During installation of the server, open Xwin Server in Windows, and “export DISPLAY=:0.0” before running cmake-gui per setup instructions.

Ensure thread_INC and thread_LIB point to the correct folders using the cmake-gui; most of the available options were left unchecked.

Libcrypt-devel is a required package for Cygwin.

The following lines must be included in fgms.conf:

```
server.name = yourservername
server.address = 127.0.0.1
server.port = 5000
server.telnet_port = 0
server.playerexpires = 10
server.logfile = fgms.log
server.tracked = false
server.daemon = false
server.is_hub = true
relay.port = 5001
relay.port = 5004
relay.port = 5003
```

To run the FGMS server in Cygwin on opening a terminal:

```
cd /cygdrive/c/cygwin/home/YourUserName/build-fgms
 ./fgms -c ./test.conf
```

REPORT DOCUMENTATION PAGE		*Form Approved* *OMB No. 074-0188*

1. REPORT DATE *(DD-MM-YYYY)* 22-03-2018	**2. REPORT TYPE** Master's Thesis	**3. DATES COVERED** *(From – To)* March 2017 – March 2018

TITLE AND SUBTITLE Design and Test of a UAV Swarm Architecture over a Mesh Ad-hoc Network	**5a. CONTRACT NUMBER**
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S) Allen, Timothy J., Captain, USAF	**5d. PROJECT NUMBER**
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAMES(S) AND ADDRESS(S) Air Force Institute of Technology Graduate School of Engineering and Management (AFIT/ENV) 2950 Hobson Way, Building 640 WPAFB OH 45433-8865	**8. PERFORMING ORGANIZATION REPORT NUMBER** AFIT-ENV-MS-18-M-172
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Intentionally Left Blank	**10. SPONSOR/MONITOR'S ACRONYM(S)**
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
DISTRIBUTION STATEMENT A. APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

13. SUPPLEMENTARY NOTES

14. ABSTRACT
The purpose of this research was to develop a testable swarm architecture such that the swarm of UAVs collaborate as a team rather than acting as several independent vehicles. Commercial-off-the-shelf (COTS) components were used as they were low-cost, readily available, and previously proven to work with at least two networked UAVs.

Initial testing was performed via software-in-the-loop (SITL) demonstrating swarming of three simulated multirotor aircraft, then transitioned to real hardware. The architecture was then tested in an outdoor nylon netting enclosure. Command and control (C2) was provided by software implementing an enhanced version of Reynolds' flocking rules via an onboard companion computer, and UDP multicast messages over a W-Fi mesh ad-hoc network. Experimental results indicate a standard deviation between vehicles of two meters or less, at airspeeds up to two meters per second. This aligns with navigation instrumentation error, permitting safe operation of multiple vehicles within five meters of each other. Qualitative observations indicate this architecture is robust enough to handle more aircraft, pass additional sensor data, and incorporate different swarming algorithms and missions.

15. SUBJECT TERMS
UAV UAS swarm architecture mesh ad-hoc network LCM UDP multicast SITL multirotor

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON** Dr. John M. Colombi, AFIT/ENV
a. REPORT U	**b. ABSTRACT** U	**c. THIS PAGE** U	UU	123	**19b. TELEPHONE NUMBER** *(Include area code)* (937) 255-3636, ext 3347 john.colombi@afit.edu