3-23-2018

Passive Radiolocation of IEEE 802.11 Emitters using Directional Antennae

Bradford E. Law

PASSIVE RADIOLOCATION OF IEEE 802.11 EMITTERS USING DIRECTIONAL ANTENNAE

THESIS

Bradford E. Law, Capt, USAF

AFIT-ENG-MS-18-M-040

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT-ENG-MS-18-M-040

PASSIVE RADIOLOCATION OF IEEE 802.11 EMITTERS USING DIRECTIONAL ANTENNAE

THESIS

Presented to the Faculty

Department of Electrical and Computer Engineering

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 Cyber Operations

Bradford E. Law, B.S.E.E.

Capt, USAF

March 2018

AFIT-ENG-MS-18-M-040

PASSIVE RADIOLOCATION OF IEEE 802.11 EMITTERS USING DIRECTIONAL ANTENNAE

THESIS

Bradford E. Law, B.S.E.E.
Capt, USAF

Committee Membership:

Barry E. Mullins, Ph.D., P.E.
Chair

Timothy H. Lacey, Ph.D., CISSP
Member

Robert F. Mills, Ph.D.
Member

AFIT-ENG-MS-18-M-040

Abstract

Low-cost commodity hardware and cheaper, more capable consumer-grade drones make the threat of home-made, inexpensive drone-mounted wireless attack platforms (DWAPs) greater than ever. Fences and physical security do little to impede a drone from approaching private, commercial, or government wireless access points (WAPs) and conducting wireless attacks. At the same time, unmanned aerial vehicles (UAVs) present a valuable tool for network defenders conducting site surveys and emulating threats.

These platforms present near-term dangers and opportunities for corporations and governments. Despite the vast leaps in technology these capabilities represent, UAVs are noisy and consequently difficult to conceal as they approach a potential target; stealth is a valuable asset to an attacker. Using a directional antenna instead of the typical omnidirectional antenna would significantly increase the distance from which a DWAP may conduct attacks and would improve their stealthiness and overall effectiveness.

This research seeks to investigate the possibility of using directional antennae on DWAPs by resolving issues inhibiting directional antennae use on consumer and hobbyist drone platforms. This research presents the hypothesis that a DWAP equipped with a directional antenna can predict bearings and locations of WAPs within an acceptable margin of error.

A ground-based hardware prototype is constructed to test this hypothesis by emulating an airborne UAV platform. `localizer`, a framework written in Python, is built to manage synchronous control of the data capture process to enable directed capture of data that is used to optimize radiolocation techniques. This data is analyzed and used to determine optimal capture parameters for predicting WAP bearing and location. Using these values, data is captured using the prototype and `localizer` framework to produce data sets for analysis.

The data captured is analyzed and bearing prediction error rates are reviewed for different interpolation techniques. The optimal interpolation technique, Piecewise Cubic Hermite Interpolating Polynomial (PCHIP), produces a median bearing prediction error of less than 14°. This research uses a least-squares optimization of multiple bearing predictions (rays) to predict the location of a given WAP over millions of combinations of real data sets. The location prediction performance is less accurate than expected, with a median error of more than 60 m; an in-depth analysis of these results is presented.

Using a directional antenna on a UAV brings distinct advantages. This research identifies a viable way for an airborne DWAP to scan, identify, and locate WAPs from a safe distance, maintaining operational stealth while performing computer network operations (CNO).

Acknowledgements

To my sweetheart, our beloved children, and above all, my Creator,
to all who so abundantly bless me, thank you.

Table of Contents

Page

Page

List of Figures

Figure Page

List of Tables

List of Terms and Abbreviations

AOA Angle of Arrival

A technique used to measure the direction of an emitter by measuring the signal arrival across elements of an antenna array.

BSSID Basic Service Set Identifier

A unique media access control (MAC) address that identifies the source access point or router for the wireless network.

BO Beacons Observed

The number of beacons observed during a single capture.

BPS Beacons per Second

The rate at which beacons are observed during a particular capture.

BE Bearing Error

The difference between true bearing (TB) and peak RSSI (PR).

CD Capture Duration

The length of time that a capture is performed.

CO Capture Overhead

The amount of time overhead necessary to conduct a capture.

CPO Capture Processing Overhead

The amount of time necessary to process a capture and generate bearing predictions for any observed wireless access point (WAP).

CW Capture Width

The number of degrees the antenna is rotated during a particular capture.

CHD Channel Hop Distance

The number of channels to skip when channel hopping.

CHI Channel Hop Interval

The amount of time to wait before hopping to the next channel.

CNA Computer Network Attack

Attacking a network in an attempt to disrupt, deny, degrade, or destroy information or connected systems.

CNE Computer Network Exploitation

Any action taken to gain unauthorized access to networked systems in order to gather intelligence.

CNO Computer Network Operations

Actions taken against a target computer system, including computer network exploitation (CNE) and computer network attack (CNA).

dBi Decibel-Isotropic

A logarithmic measurement of forward antenna gain relative to a reference hypothetical isotropic antenna.

dBm Decibel-Milliwatt

A logarithmic measurement of power ratio to a reference value of 1 mW

DT Detection Time

The amount of overall time necessary to capture data.

DWAP Drone-Mounted Wireless Attack Platform

A wireless attack platform built on a unmanned aerial vehicle (UAV), potentially composed of low-cost consumer-grade hardware and free open source software (OSS) software.

FCRR Focused Capture Rotation Rate

The rotation rate (RR) used when conducting a focused capture.

FCW Focused Capture Width

The capture width (CW) used when conducting a focused capture.

GPIO General Purpose Input and Output

An array of specialized input and output pins present on the Raspberry Pi that allows analog and digital signaling to external devices.

GPS Global Positioning System

A global system of satellites that enables precise navigational and surveying facility.

IB Initial Bearing

The bearing that the antenna is facing when a capture begins.

LE Location Error

The distance between predicted position (PP) and TP.

mW Milliwatt

Unit of measurement for power of received signal strength indication (RSSI).

OSS Open Source Software

Free computer software with freely-available source code.

PR Peak RSSI

The bearing where the highest received signal strength indication (RSSI) is recorded.

PP Predicted Position

The position of a wireless access point (WAP) that is predicted by the `localizer` framework.

PWM Pulse Width Modulation

A modulation technique used to control the speed of the stepper motor via the stepper motor controller.

RSSI Received Signal Strength Indication

A measure of the energy observed by an antenna when receiving a signal.

RR Rotation Rate

The antenna rate of rotation during a capture.

SSH Secure Shell

Secure communication protocol that is used to connect to the prototype.

SSID Service Set Identifier

A sequence of characters that uniquely names a wireless local area network; a wireless local area network name.

SNR Signal-To-Noise Ratio

A measure that compares the level of a desired signal to the level of background noise.

TDOA Time Difference of Arrival

A technique used to measure the location of an emitter by comparing the times a signal is received by multiple, synchronized receivers.

TOA Time of Arrival

A technique used to measure the distance of an emitter by comparing the time it takes for the signal to be received by a synchronized receiver.

TU Time Unit

A unit of time equal to 1024 microseconds introduced in the IEEE 802.11 standard. 802.11 standard sets the beacon rate at one beacon every 100 TU, often rounded to 10 beacons per second (BPS).

TB True Bearing

The real bearing to the wireless access point (WAP).

TP True Position

The true position of a wireless access point (WAP).

UAV Unmanned Aerial Vehicle

An aerial vehicle that is either controlled remotely or autonomously.

vFCW Virtual Focused Capture Width

A focused capture width (FCW) derived from a wider capture width (CW) that is used to determine the optimal FCW.

WAP Wireless Access Point

An IEEE 802.11 hardware device that serves as a node on a local area network and allows wireless access.

PASSIVE RADIOLOCATION OF IEEE 802.11 EMITTERS USING DIRECTIONAL ANTENNAE

I. Introduction

1.1 Background

Attacks on Wi-Fi networks have grown in tandem with Wi-Fi growth and adoption, among private, government, and military organizations alike. Even though attacks on Wi-Fi networks may be conducted remotely, they often require relativity close proximity to the target, as little as 33 meters for some protocols. Physical security (i.e., fences, security guards) may also significantly increase the difficulty of wireless attack by forcing an attacker to approach near enough to be detected.

Consider also the growing availability of low-cost unmanned aerial vehicles (UAVs), such as fixed pitch multi-rotor helicopters (quadrocopters) and commodity hardware. This combination has created a new wireless attack vector in the form of drone-mounted wireless attack platforms (DWAPs). A substantial advantage of this platform is that it insulates an attacker from discovery, since he may control the drone from miles away using cheap mobile broadband. Private and government organizations also have an interest in the potential of these DWAPs as they seek to understand better what threatens their network security. Threat emulation, the doctrine of defenders emulating real-world adversarial threats as they conduct readiness exercises, necessitates the development of offensive security for defensive purposes. Using drones to attack wireless networks is a real threat and is growing; network defenders necessarily need to understand and emulate the threat to adequately defend against it.

1.2 Problem Statement

This research is limited in scope to low-cost consumer-grade hardware and open source software (OSS). Attack platforms built on this type of hardware typically suffer from the unique disadvantages of being relatively loud and using low-gain omni-directional wireless antennae. A result of this combination is that under normal conditions, a DWAP that uses omnidirectional antennae is audible before it is within range to conduct an attack. Consider an attacker taking a commercial hobbyist drone close to a secure facility to attack its networks. The attacker would have to get very close to conduct the attack. A typical drone emits around 76 dB and would be audible within 100 meters [1]. Stealth is invaluable for an attacker, and he loses it using traditional wireless attack techniques.

Omni-directional antennae are not ideal for long-range wireless attacks. On the other hand, directional antennae have been in service for years conducting wireless attacks of many kinds, since they are tuned to focus the transmission and reception of signals in a particular area, significantly increasing transmission and receiving range.

While utilizing directional antennae on a hobbyist-grade drone solves some of the disadvantages mentioned earlier, it also introduces new problems that must be addressed for it to be effectively used in conducting wireless attacks. Some of the difficulties that this research seeks to surmount include finding the right bearing to aim the antenna to maximize both signal-to-noise ratio (SNR) and standoff distance, which is the distance between an attacker and the target and should be maximized to decrease chances of detection. Another unique problem for directional antennae in this context is surveying the surrounding area for potential targets. A target sweep is conducted differently using a directional antenna than a standard omni-directional antenna.

Furthermore, if autonomy is desired, a robust system for locating near and distant targets is necessary; overcoming these obstacles is even more critical for autonomous DWAPs.

1.3 Research Goals

With the possible advantages of using directional antennae, this research seeks to overcome the obstacles that accompany directional antennae. Accurately locating the direction of a wireless signal is of primary importance, so that the DWAP may conduct its attack. This work intends to determine the efficacy of the proposed localization method by measuring the median bearing and location errors to all experiment wireless access points (WAPs) from multiple capture locations.

This research also presents a software project `localizer` that serves as a framework for capturing research data, as well as performing live target bearing and location determination. This software package has three primary roles, namely, batch data capture, real-time data capture, and data processing. The first role, batch data capture, is used to conduct all mass data collection for research and analysis. The second role, real-time interactive capture, is used to demonstrate the capabilities of the platform and is the primary mode used when conducting simulated attacks, including from a DWAP. The third role is used to process captured data sets for future investigation. The software is easily extended to meet the needs of future functions and research.

When used for real-time network attack, the actual target position is not the primary interest for a DWAP; the attacker just needs to know which bearing to direct the antenna, or which direction to move to increase the received signal strength indication (RSSI). In scenarios of network mapping, however, the location of the target is desired and may provide valuable intelligence about the target. This research intends to move beyond predicting a target's bearing and identify an optimal approach using sets of bearing predictions to predict a target's position.

1.4 Hypothesis

This research hypothesizes that a DWAP-mounted directional antenna may be used to identify the bearing and location of a WAP within an acceptable margin of error. A WAP can be discovered by a DWAP passively capturing beacons during an initial *wide* capture, where the monitoring channel is changed regularly to discover all broadcasting WAPs within range. During a wide capture, each beacon that is received is grouped by basic service set identifier (BSSID). Each beacon in a group is considered a single data point, consisting of two primary properties, bearing and RSSI.

Bearing prediction accuracy is expected to improve by performing interpolation on the sparse beacon data that creates a continuous map of the RSSI as a function of bearing.

The DWAP may conduct a *focused* capture, with the channel held constant to that of the targeted WAP, which ensures no beacons are missed due to monitoring a different channel. This higher beacon capture rate is expected to provide improved bearing prediction accuracy.

Bearing Discovery Method. This research hypothesizes that at standard beacon rates of one beacon every 100 TU, equivalent to one beacon every 102.4 ms, and using optimized capture parameters, a DWAP can identify WAPs within range and determine their respective bearings. The predicted bearings are expected to be accurate within 45° from true bearing in a wide capture (using channel hopping), and within 15° from true bearing using a focused capture (constant channel) width of 60° or less.

Location Discovery Method. Additionally, this research hypothesizes that the *locations* of WAPs can be discovered using multiple bearing predictions using least-squares optimization to find the location that is closest to each bearing prediction. These coordinates are presumed to be the optimal location prediction based on the provided data and is hypothesized to predict with an accuracy of less than 10 m of error from truth.

1.4.1 Hypothesized Capture Method.

The list below outlines the hypothesized method to capture data and predict observed WAP bearings.

1. **Locate an Ideal Location to Perform Capture.** An ideal location is one free of immediate obstructions between the prototype and potential emitters.

2. **Initiate Wide Capture.** Begin a 360° sweep, changing the monitored channel regularly to ensure complete coverage of all channels.

3. **Process Results.** For each observed emitter, use an optimal interpolation method to fill in the missing data and predict the emitter's relative bearing.

4. **Select a Target.** Of the observed emitters, an operator or operational program selects a target for a focused capture.

5. **Optional: Initiate Focused Capture.** If a more accurate bearing is necessary or desired, match the prototype capture channel to the target emitter's channel and perform a focused sweep centered on the predicted bearing for the target emitter.

6. **Optional: Process Results.** If a focused capture is performed, the data is processed and a prediction is made using the interpolated capture results.

7. **Initiate Action.** Now that a reasonably sound bearing is determined for a target, the operator may choose to connect to the WAP, conduct computer network operations (CNO) on it, or proceed to another location to capture more data.

1.5 Approach

Equipment. A ground-based prototype of a DWAP is designed and constructed consisting of a directional antenna, motor, wireless interface, power source, and processor to execute the `localizer` software package, which is responsible for data capture and bearing and location prediction.

Parameter Discovery. Ideal parameters such as antenna rotation speed and channel hop rate are discovered by capturing data with the prototype hardware. After sufficient data has been captured for each parameter under test, the parameters' respective beacon observation rates are compared and optimal values for each parameter are identified.

Data Capture. Once the parameters are determined, data captures are conducted repeatedly at those values until a sufficiently large data set has been collected. Different methods of capture are also performed, such as wide captures and focused captures.

Analysis. Captured data is analyzed to determine the bearing and location prediction errors using the hypothesized methods. An analysis is performed on the results to provide insight into the findings.

1.6 Assumptions and Limitations

This research is conducted under the following understood assumptions and limitations, namely:

- **Environmental Interference.** Objects between the emitter and receiver are assumed to be typical commercial building material and not vary significantly between samples. The experiment location represents a "worst case" reflection environment on the ground-level when compared to the reflections of an airborne environment. In other words, the buildings surrounding the experiment site are assumed to contribute to unwanted reflections significantly more so than the non-reflective free space surrounding an airborne DWAP. With this in mind, the `localizer` performance is expected to improve dramatically when the system is deployed on a DWAP.

- **Prototype Interference.** Any electromagnetic interference generated by the prototype, such as from signal wires, power sources, or stepper motor windings, is considered non-destructive in the 2.4 GHz frequency range and is ignored.

- **Weather Effects.** The influence of weather (such as humidity) on the behavior of beacon frames is assumed to be minimal and is ignored.

- **Bearing Consistency.** The RSSI from any WAP is assumed to have maximum relative intensity in the same direction as the WAP. In other words, the bearing of the strongest RSSI is the direct bearing to the WAP. This assumption ignores reflections that may cause the stronger RSSI readings from incorrect bearings. In practice, a DWAP need only locate the bearing with the maximum RSSI.

- **Beacon Sufficiency.** The captured packets in this research and experiments are limited to beacon packets. This work is primarily interested in optimizing the process of locating a static emitter's bearing and location; this limitation serves to simplify the experiment treatments, environment, data processing, and data analysis. Future research could incorporate active listening, including all possible signal localization sources.

1.7 Contributions

This thesis contributes to the body of DWAP research, specifically wireless network localization. It presents a solution to an unavoidable problem when localizing WAPs from a UAV using a directional antenna, and shows empirically that the proposed method of directional radiolocation can predict emitter bearing for use on future DWAPs.

1.8 Thesis Overview

This thesis is arranged in six chapters. Chapter II presents fundamental concepts of radiolocation as well as support applications of radiolocating UAVs and attack application of DWAPs. Chapter III discusses prototype design and focuses on hardware and software composition separately. Chapter IV presents the experiment methodology, including the standard parameters, metrics, and testing process. Chapter V reviews the results of the collected data. Finally, Chapter VI summarizes the research and discusses opportunities for extensions to this research.

II. Background and Related Research

2.1 Overview

This chapter discusses the necessary radiolocation background details and surveys applications of drone-based localization Wi-Fi systems. Relevant computer network exploitation (CNE) and computer network attack (CNA) applications are also covered.

Section 2.2 discusses radio performance of omnidirectional and directional antennae. Section 2.3 covers radiolocation principles, including basic properties such as received signal strength indication (RSSI), time of arrival (TOA), and angle of arrival (AOA). This section also covers relevant radiolocation principles such as triangulation, trilateration, the weighted-centroid algorithm and probabilistic-based algorithms used in predicting an emitter's location. Section 2.4 discusses applications of UAV-based radiolocation, such as wardroning, emergency response, and user localizing. It also covers possible DWAP attack vectors. Section 2.5 reviews sparse data sets and interpolation techniques that can fill in gaps in data sets.

2.2 Radio Performance Comparisons

One of the most significant characteristics of an antenna is its gain. Antenna gain is measured in decibel-isotropic (dBi) and is a log ratio relative to the hypothetical isotropic antenna, an antenna that uniformly distributes energy from a point in all directions. Increases in dBi indicate that the energy is focused in a particular direction, plane, or is otherwise not uniformly distributed.

2.2.1 Omnidirectional Antennae.

Omnidirectional antennae used in Wi-Fi applications are typically small, quarter wavelength dipole antennae with typical dBi values between 3 and 6 [2], although values approaching and exceeding 12 dBi are claimed by some antenna manufacturers [3]. These antennae have a disc radiation pattern expanding perpendicular to the antenna in 360°. As the gain increases, the pattern stretches horizontally and flattens vertically. Figure 1 shows the radiation pattern for a typical dipole antenna.

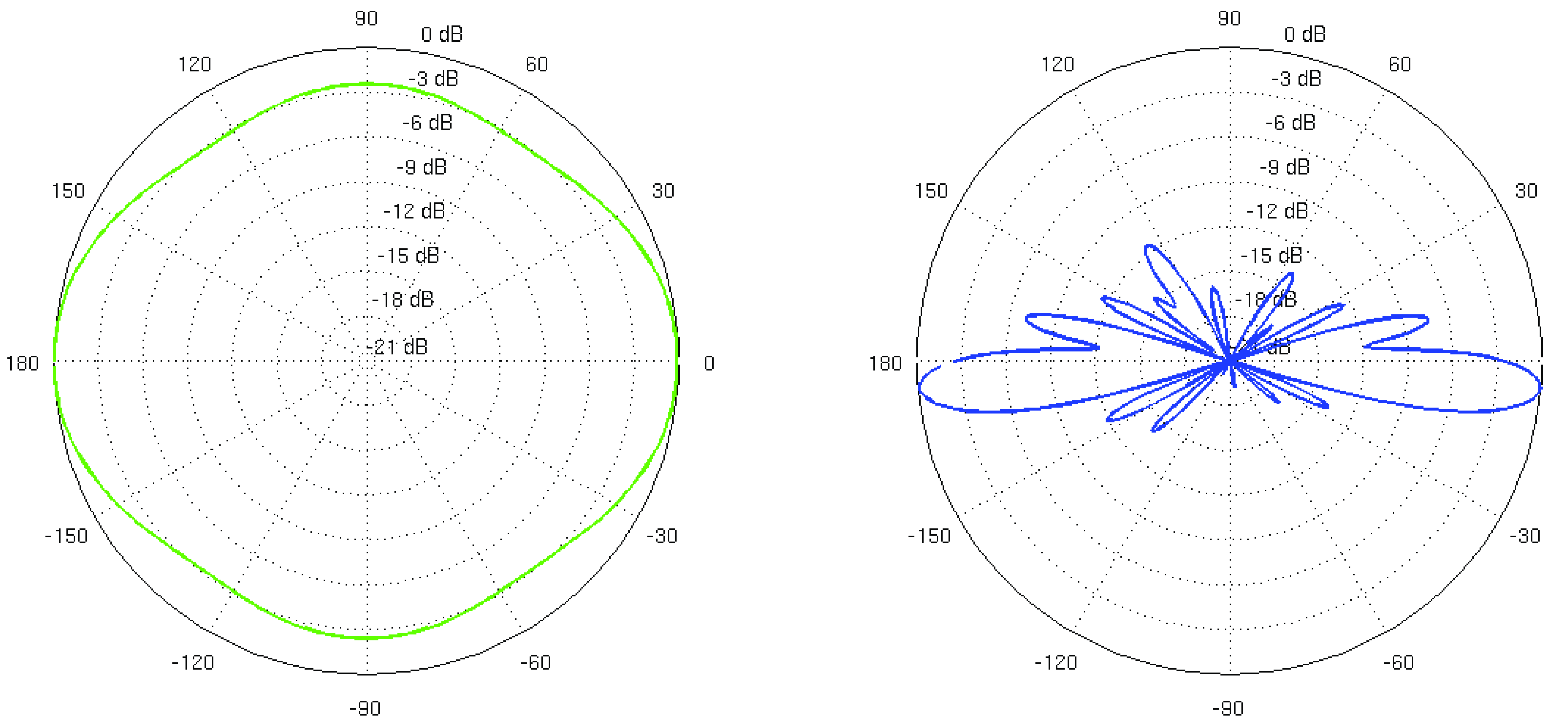
2.2.2 Directional Antennae.

Directional antennae have a directed radiation pattern, where the azimuth and elevation planes are directed in some manner, such as shown in Figure 2. Directional antennae are able to claim higher dBi because they are able to project electromagnetic radiation much farther, and are likewise more sensitive to receiving signals from longer distances.

Using a directional antenna provides two significant advantages for this research: namely, increased range and directionality feedback.

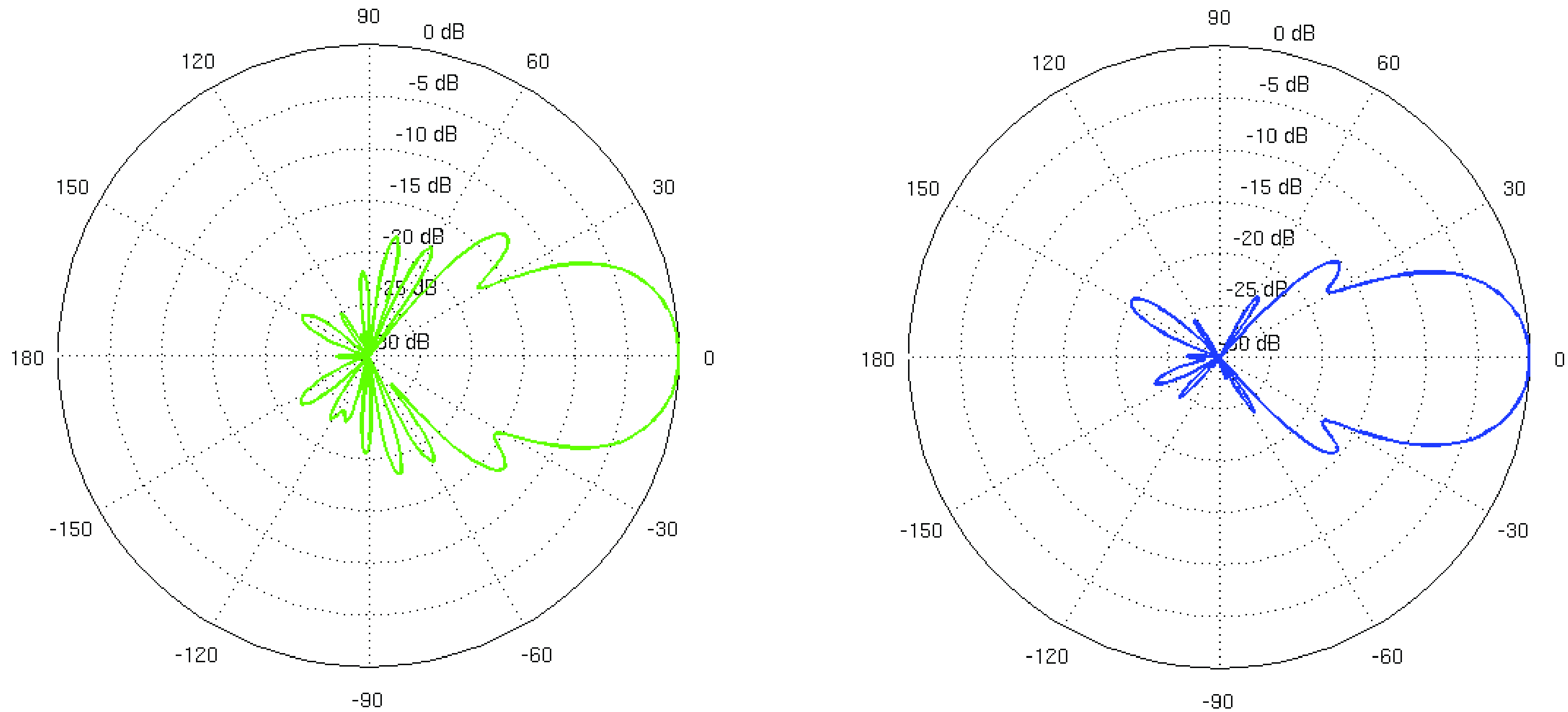
2.3 Radiolocation

Radiolocation is the process of determining the position, velocity, and other characteristics of an object by analyzing the propagation properties of radio waves [4]. This process includes, for example, measuring the reflected (backscattered) signals of radar or locating an emitter by using multiples receivers to passively analyze that emitter's signals. This research is primarily concerned with the latter method, specifically, determining the location of 802.11 emitters by passively analyzing their beacon emissions, and any further use of the term *radiolocation* in this work is limited to this application.



(a) Dipole Azimuth Plane Pattern **(b) Dipole Elevation Plane Pattern**

Figure 1. Dipole Radiation Pattern [3]



(a) Yagi Azimuth Plane Pattern

(b) Yagi Elevation Plane Pattern

Figure 2. Yagi Radiation Pattern [5]

The radiolocation methods discussed here are computationally simple, and ideal for low-powered hardware, but are vulnerable to interference (e.g., attenuation, reflections, and multipath propagation) from objects in or around the signal path. In other words, objects between the emitter and receiver, as well as reflective surfaces surrounding them, reduce the accuracy of these methods. For airborne DWAPs, this disadvantage is mitigated somewhat by the low reflectivity of the sky at the 2.4 GHz and 5 GHz Wi-Fi frequencies.

There are many methods to perform radiolocation, however many of them require multiple receivers or specialized antenna arrays. This section reviews fundamental properties of radio communications, as well as advanced and straightforward radiolocation techniques. Each technique is broken down by the type of localization it provides, such as bearing only, distance only, or location. Additionally, Wi-Fi specific properties are discussed as well.

2.3.1 Received Signal Strength Indication.

RSSI is a measurement of the amount of energy that a receiving antenna observes. RSSI is measured in decibel-milliwatts (dBms) and is very useful in this research as a measurement point of reference. As dBm increases or decreases from zero, the relative ratio that it measures grows exponentially.

The milliwatt (mW) is a measurement of received power and is directly related to the metric RSSI by the equation:

$$mW = 10^{\frac{dBm}{10}} \tag{1}$$

The relationship between RSSI (in dBms) and mWs is shown in Figure 3, where linear increases in dBms produces exponential increases in mWs, and a value of 0 dBm is equivalent to 1 mW.

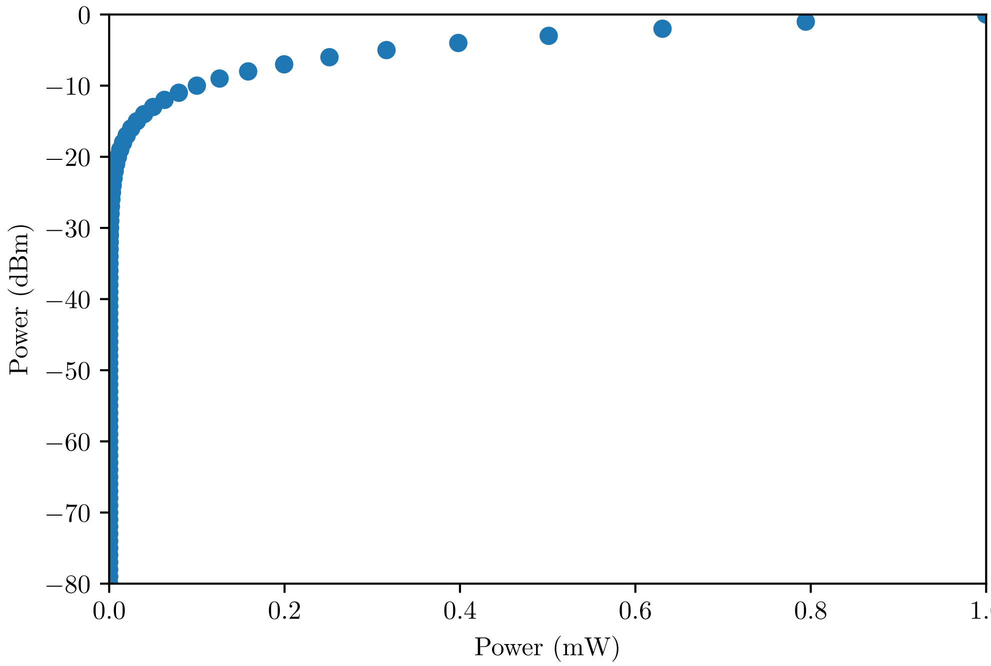


Figure 3. Decibel-Milliwatt to Milliwatt Scale

Distance. The signal's RSSI can help determine the emitter's distance if the originating broadcast power is known. If the originator's signal strength is known, that value may be used with the observed RSSI to determine the attenuation of the signal. Additionally, if the propagation characteristics of the medium between the transmitter and the receiver are known, the distance between the emitter and receiver may be determined.

Bearing. A directional antenna or omnidirectional antenna array may be used to collect observations of RSSI that predict the bearing of the emitter. This research uses a directional antenna to collect observations of RSSI as a function of bearing to determine the best predicted WAP bearing; the technique is discussed at length in Chapter IV.

Location. If multiple receiver readings can be observed at different locations, that data can be used to trilaterate the signals and predict the source signal location. Because this approach requires knowledge of the source signal strength, it is of limited use in this research that which assumes no transmission power data is encoded in the packets.

2.3.2 Time of Arrival.

Time of arrival (TOA) is the measure of the time a signal is received. This information can help determine emitter location under certain circumstances.

Distance. If the time of signal emission is known, TOA may use signal propagation duration to determine emitter distance. TOA is the amount of time elapsed between when the signal is sent and when it arrives at a receiver. The clocks in the emitter and receiver must be synchronized, and the propagation characteristics of the medium between the transmitter and the receiver must be known to accurately estimate the emitter distance. Consider Figure 4, where an emitter broadcasts a packet with an encoded timestamp synchronized to a common clock such as global positioning system (GPS) satellites. A single receiver synchronized to the same clock may use its known location and the TOA to determine the emitter's distance.

Location. As with RSSI, measurements taken at multiple points may allow the receiver to trilaterate the signal to predict the emitter's location. If two receivers are used, as shown in Figure 4, the location can also be determined much in the same way as the distance.

TOA requires prior knowledge of the time of signal transmission from the transmitter and synchronized clocks. Furthermore, TOA is vulnerable to multipath errors; reflections may interfere with the perceived signal time of arrival and hinder accurate emitter location prediction.

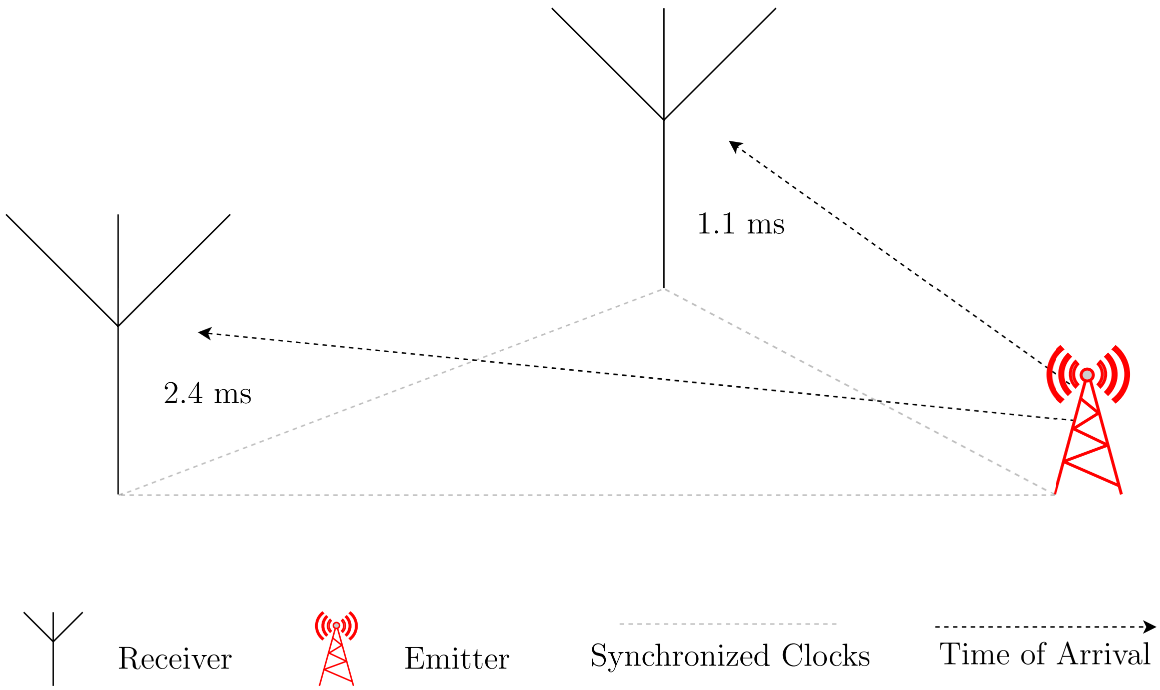


Figure 4. Time of Arrival - One Receiver for Distance, Two Receivers for Location

2.3.3 Time Difference of Arrival.

Time difference of arrival (TDOA), or multilateration, differs significantly from TOA, in that it does not require synchronized clocks between the receiver and emitter. Instead, it relies on synchronized clocks between multiple receivers. Pairs of receivers measure the duration that a signal takes to pass between them and may use this information to determine emitter distance.

Distance. A pair of synchronized receivers may determine a probable distance to the emitter by comparing their relative distance from each other with the time and RSSI differences from their observations.

Location. The observations of a single pair of receivers can be used to generate a hyperbolic curve of possible emitter locations. More pairs of receivers may do the same, generating additional hyperbolic curves. These curves intersect at the emitter's probable locations.

2.3.4 Angle of Arrival.

Observing the angle of a received signal, relative to some chosen reference angle such as magnetic north, is an inexpensive and simple way to determine the direction of an emitter.

Direction. The use of a directional antenna (or antenna array) allows for the determination of the angle of the signal's origin. A single directional antenna can determine an emitter's direction by rotating and observing the RSSI as antenna direction changes - this is the method developed in this research to predict WAP bearings. A specialized array of antennae may also determine signal direction by observing differences in RSSI over many small, discrete antennae.

Location. Multiple measures of angle of arrival (AOA) may be collected at different locations and combined to predict the emitter location using triangulation. A single receiver may also be used, if the emitter is stationary, to predict the emitter location by determining the AOA at different coordinates. Furthermore, multiple synchronized receivers may take measurements over time to track moving emitters. Figure 5 demonstrates this principle; the emitter in the center is observed by three receivers. Each determines an AOA, and triangulation is trivial using the combined data to determine the emitter's location.

As with using AOA to determine an emitters direction, this research uses AOA readings from multiple positions to predict stationary source signal locations.

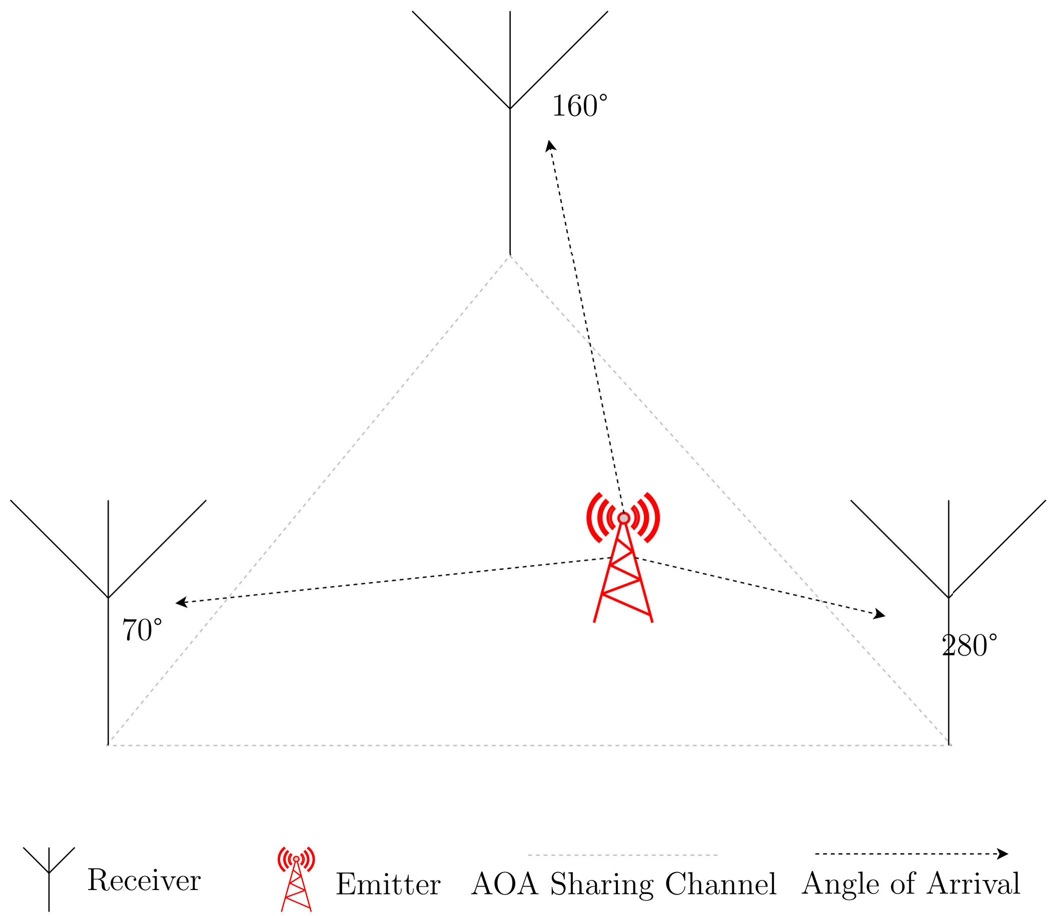


Figure 5. Angle of Arrival - Multiple Receivers Determine Emitter Location

More advanced techniques use some or none of the above signal attributes to improve the prediction accuracy of emitter location. Two of the most used techniques [6] are weighted-centroid-based algorithms and probabilistic-based algorithms.

2.3.5 Triangulation.

At its core, triangulation is the use of *angles* to determine an object's location. If two observers with a known position are observing an object with an unknown position, the two observers can share their observation angles and determine the object's location by forming a triangle with the angles and their two known positions.

Triangulation may also be performed by a single observer making multiple observations of a single stationary object. This technique is used later in this research to derive an emitter's location coordinates.

2.3.6 Trilateration.

Trilateration may be used to determine an object's location by measuring the distance to an object. Two observers may determine an object's distance from them without knowing the object's angle (such as when using an omnidirectional antenna and deriving a distance from RSSI). This distance may be considered a radius for a circle, or sphere if the object is not restricted to a plane. Two observers then form two circles which intersects at two points, either of which may be the object's location. A third observer may further narrow the object's location to a single area or point.

This technique is effective at locating emitters within a margin of error as low as 30 m [7], however lower margins of error are difficult when using RSSI to determine distance due to low attenuation over clear space, and high attenuation through physical materials.

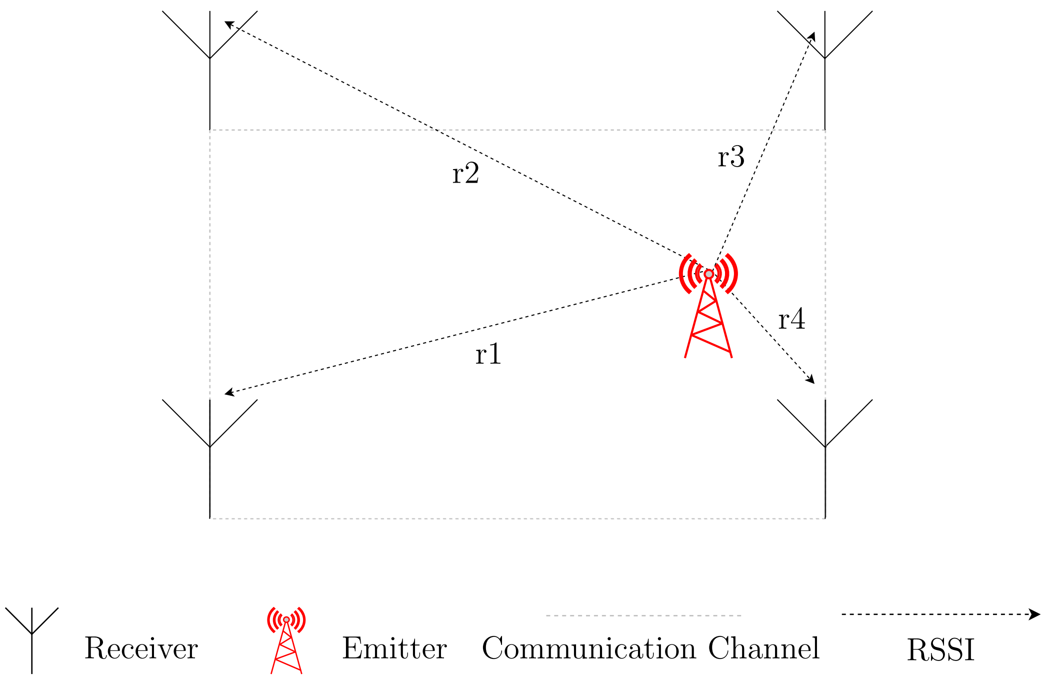
2.3.7 Weighted-Centroid-Based Algorithms.

Weighted-centroid-based algorithms estimate an emitter's location by calculating the arithmetic mean of each observation. The accuracy is improved by weighting each observation by the RSSI that is observed for that value. More distant observations (with a lower RSSI) do, therefore, affect the predicted location less than an observation that is closer (with a higher RSSI). This effect is particularly strong when using mW instead of dBm, since a linear reduction of dBm is equivalent to an exponential reduction in mW (see Figure 3).

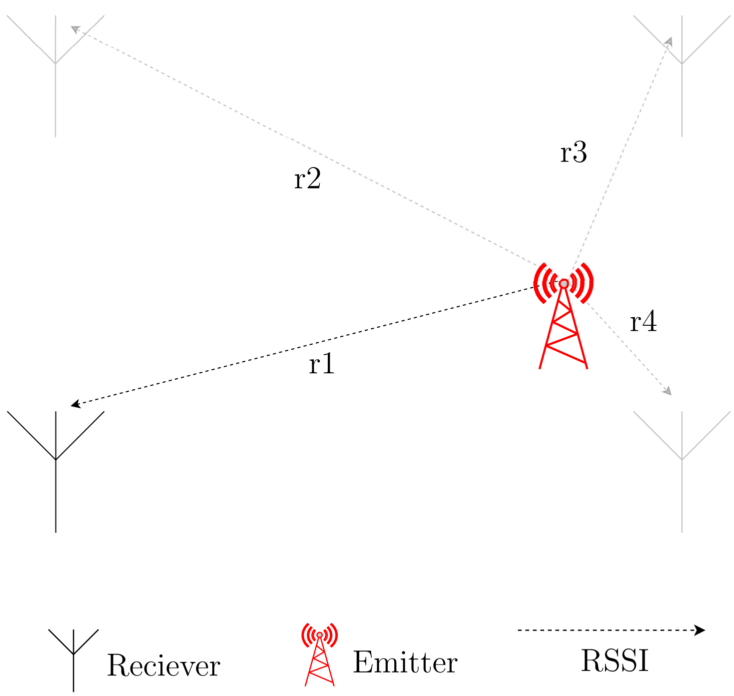
Figure 6a demonstrates this concept with four connected receivers that measure RSSI from a single emitter. The receivers share their observations `r1`, `r2`, `r3`, and `r4` and locate the emitter. Figure 6b shows an alternative approach that uses a single mobile receiver to make multiple observations RSSI; if the emitter remains stationary, a single mobile receiver may locate an emitter.

2.3.8 Probabilistic-Based Algorithms.

A probabilistic-based algorithm uses machine learning that avoids traditional geometric radiolocation techniques (i.e., TOA and AOA) [8]. Instead, it employs large volumes of collected data, such as position, emitters observed, and RSSI values for each emitter, to tune a machine learning algorithm. The resultant algorithm produces a map that is considerably more accurate than weighted-centroid-based algorithms [6]. The error function of the machine learning algorithm optimizes the coefficients of the prediction algorithm based on the training set data. This algorithm can be optimized by adjusting the amount of training data and variables. This approach is also known as fingerprinting, as the prior data is considered a device's "fingerprint."



(a) Multiple Receivers



(b) Single Receiver

Figure 6. Weighted-Centroid Localization Algorithm

2.3.9 Wi-Fi Principles.

Wi-Fi networks are radio networks, and therefore radiolocation of a Wi-Fi emitter can be performed by using the same principles discussed above. Additionally, Wi-Fi packets may contain information that aid in determining its location. The BSSID is a critical metric in this research.

BSSID. This 48-bit number is embedded in every packet transmitted by a WAP and uniquely identifies it. In an environment with many emitters, this serves to isolate signals from a particular target from neighboring WAPs. The BSSID is also used by nodes communicating with a WAP to identify that particular WAP as the recipient of the packet.

Channel. The channel in use by the WAP is embedded into the beacons it broadcasts. This is important, since relying strictly on the receiver to determine the channel is problematic; the channels defined by 802.11 specifications are close enough that traffic on a particular channel may be observed on adjacent channels and the receiver may attribute the wrong channel to a particular WAP. Knowing a WAP's true channel is useful for building a complete data set for bearing and prediction location.

Many commercial Wi-Fi network interface cards can be put into *monitor mode*, a mode of operation where the network interface listens to any valid signals it receives, irrespective of who the traffic is intended for. By hopping channels, a monitor mode network interface may observe traffic on all channels, although it may only monitor a single channel at a time. This mode is used to scan for nearby WAPs, record key exchanges, and eavesdrop on unencrypted communications.

On some devices, monitor mode also adds a special layer to captured data called the *Radio Tap* layer. This layer contains physical attributes of the received radio signal, including RSSI.

2.4 Radiolocation Applications

The research developed here can be applied in numerous ways, including in the following radiolocation applications.

2.4.1 Emergency Response.

Numerous applications of using a Wi-Fi equipped UAV in emergency response have been proposed [9, 10, 11]. Most smart-phones regularly emit Wi-Fi probe requests, and since smartphone adoption is nearing saturation, research has focused on localizing those probe request using a "wardroning" approach. Research has shown that Wi-Fi enabled phones can be detected from up to 200 meters away [10]. By using optimized flight paths, drones can maximize the probability of detecting an emitting phone; once detected, they may adjust the flight path to hone in on the emitting source. These platforms use omnidirectional antennae that limit the detection range and require more active flying, which increases the amount of time needed before the drone is within range of a potential search and rescue candidate.

2.4.2 Wardriving.

Wardriving is the act of locating WAPs by continuously collecting Wi-Fi beacons and mapping the point of detection and service set identifier (SSID) [12]. The term *Wardriving* developed from wardialing, the act of dialing random or consecutive phone numbers in search of modems. Wardriving began in 2000 and grew to be quite popular among amateur technology hobbyists in the following years [13]. Wardriving has been credited with increasing the security of Wi-Fi access points by exposing the great number and locations that were originally unsecured. The location of Wi-Fi access points and their security level is of concern to network defenders responsible for conducting rogue WAP audits, as well as those with malicious intent seeking unsecured networks or targeting specific individuals and organizations for network attacks.

Even though wardriving has waned in recent years [14], as shown in Figure 7, there is still a need for network administrators to accurately and quickly perform a wireless site survey. In like manner, malicious attackers and penetration testers continue to research better ways to map Wi-Fi access points, including using transportation other than cars, such as walking, bicycles, trains, and more recently, drones. Figure 8 demonstrates the results of wardriving across the United States, and Figure 9 shows a city wardriving map where green, white, and red symbols indicate unsecured, WEP encrypted, and WPA protected WAPs respectively.

Prediction Accuracy. The accuracy of Wardriving has typically been very poor. Most wardriving systems in use consist of regularly polling a GPS device while also recording which Wi-Fi access points are detected at that time. The simplest method of localizing uses the GPS coordinates at the time of detection. A slightly more advanced method uses location averaging when the same BSSID is detected at multiple locations [15], and weighted-centroid averaging would likely give an even better estimate. Research has shown progress in implementing probabilistic-based algorithms that show a significant increase in prediction accuracy [6]. This performance, however, is limited by the range and non-directionality of omnidirectional antennae.

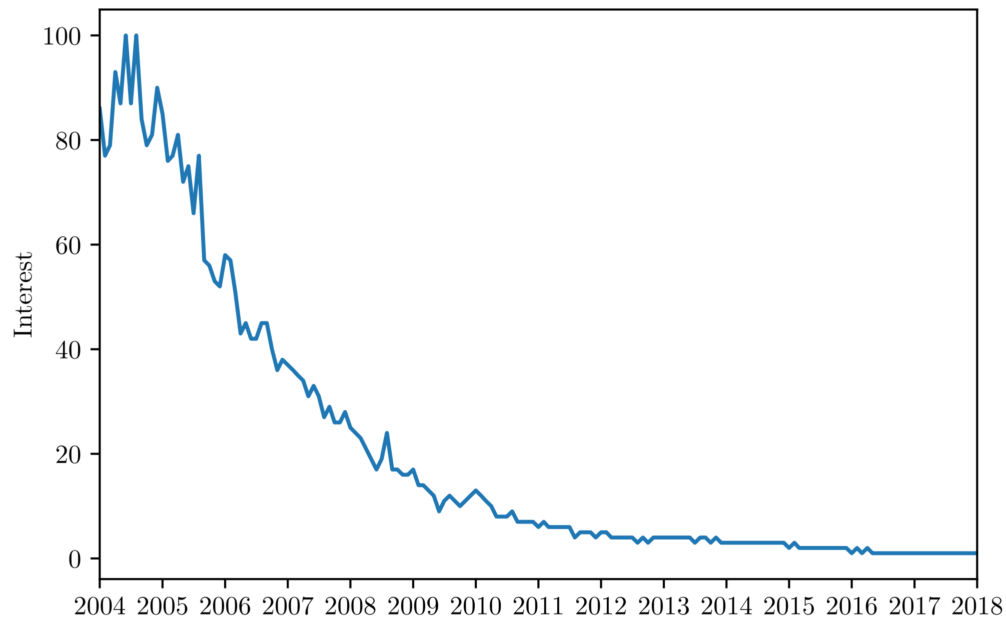


Figure 7. Wardriving Interest Relative to Peak in 2004 [14]

2.4.3 Computer Network Operations.

According to Bruce Schneier, Computer network operations (CNO) are both computer network exploitation (CNE) and computer network attack (CNA) operations. CNE is any action taken to gain unauthorized access to networked systems to gain information, while CNA is the act of attacking a network in an attempt to disrupt, deny, degrade, or destroy information or connected systems [18].

Radiolocation assists in both CNE and CNA in cases where the attack is wireless and the locations of possible points of ingress (WAPs) are not known. In taking CNE and CNA actions, maximum standoff distance is typically desired, whether operating from a DWAP or not, to reduce exposing the attacker's presence to the target. Once a target has been identified, the attack may commence using the same hardware used to locate the target WAP to connect and conduct CNO.

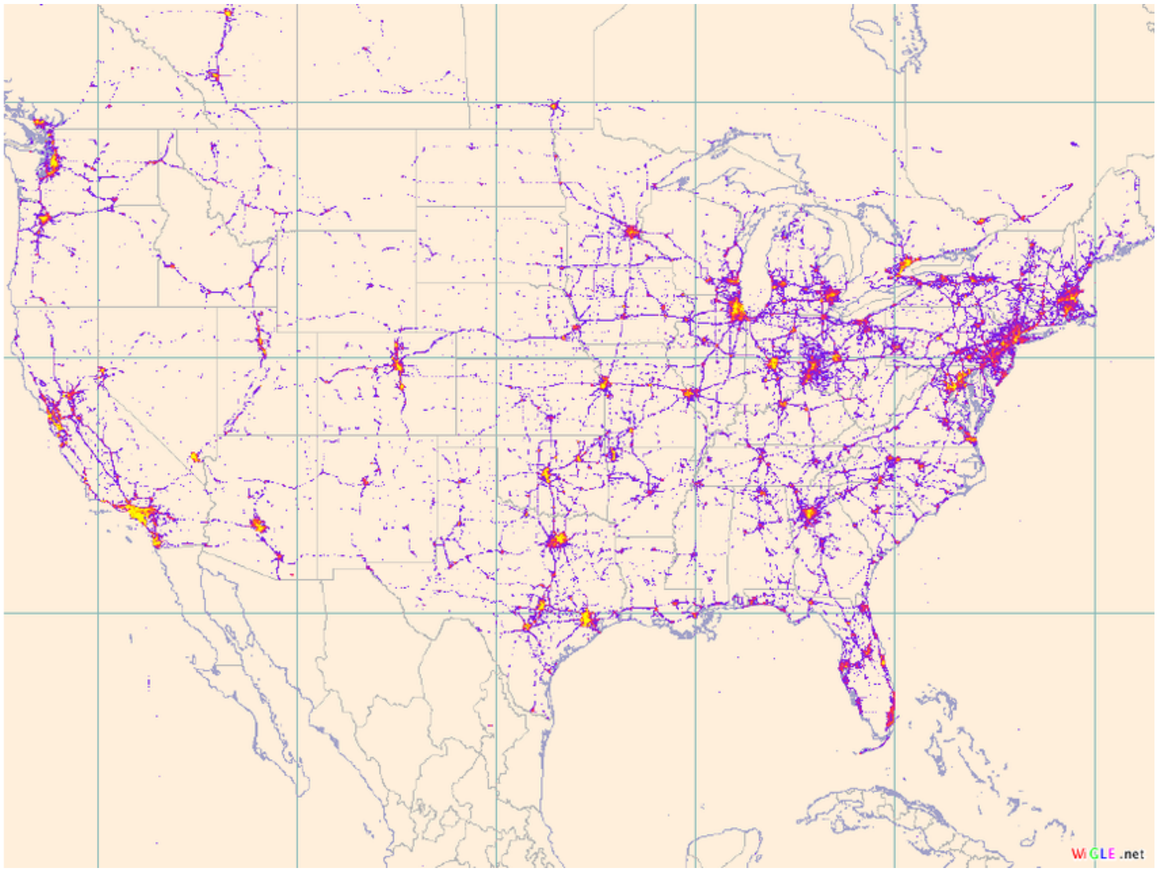


Figure 8. Map of Wi-Fi Access Points from Wardriving [16]

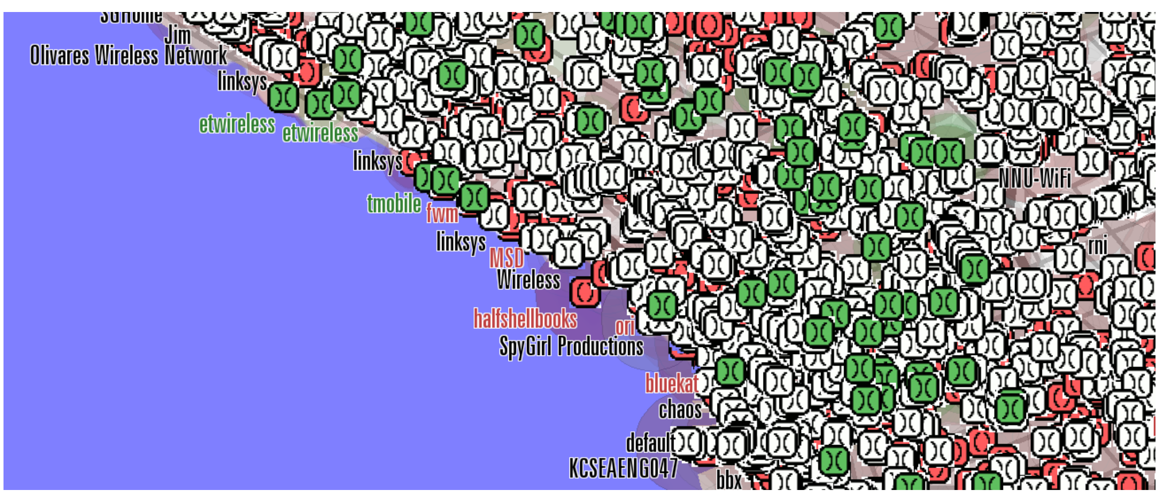


Figure 9. Detailed Map of Wi-Fi Access Points from Wardriving [17]

Man-in-the-Middle. Products like the Wi-Fi Pineapple demonstrate the attack surface any Wi-Fi-using organization presents to attackers [19]. The Wi-Fi Pineapple is capable of an attack that maliciously responds to client probe requests and bridges target client connections to the Internet to serve as a transparent proxy. As powerful as the Wi-Fi Pineapple attack suite is, its utility and area of effect increases using directional antennae; an attacker may focus the Wi-Fi Pineapple on a specific building or perform attacks from a greater distance.

Denial of Service. A directional antenna is ideal for a denial of service attack such as jamming. An attacker may either continually transmit interference signals (active jamming) or exploit weaknesses in the protocol (intelligent jamming). Contrasting with active jamming, intelligent jamming allow an attacker to produce denial of service effects without using large amounts of power [20].

User Tracking. Many smartphones, among other Wi-Fi enabled devices, continually broadcast probe requests, releasing the SSID and BSSID of the WAP they have connected to in the past, as well as their own globally unique MAC address. This information can reveal private details about a person. Personal identifying information can be gleaned from smartphone probe requests, including home address and social connections [21]. Research has also shown that building occupancy may be determined, and users may be located in search and rescue scenarios, by monitoring probe requests from the users' smartphones [22]. Also, commercial products already exploit this vulnerability to track consumers [23]. A directional antenna can significantly increase the maximum distance that user tracking could be performed.

2.5 Sparse Data Interpolation

Sparse data sets are sets of data that are not continuous, or when discrete, do not contain values at regular intervals. An example of a sparse data set is one that is created by a sensor that records readings at irregular intervals. Another relevant example is RSSI readings taken as a function of time or position, where the emitting signal was occasionally not strong enough to be detected by the receiver. Sparse data sets have gaps in the data that may need to be filled in by making an educated guess. The process of filling in the gaps is called interpolation.

There are many methods of interpolation, usually suited for specialized applications. For example, polynomial interpolation is useful for deriving a polynomial function from a series of points where the resultant function passes through all the points.

A more advanced category of interpolation techniques is spline interpolation, where each interval between pairs of data points is treated as a low-degree polynomial that smoothly transits between each interval. This has the advantage over high-degree polynomial interpolation because the low-degree polynomials are less computationally intensive to derive and evaluate.

Many more advanced interpolation techniques exist, including very specialized methods. Large libraries of interpolation techniques are a part of most data analysis packages, and are used extensively in this research to determine the optimal interpolation technique for the sparse data sets produced from these experiments.

Section 5.4.1 describes interpolation techniques that are used in this research to predict WAP bearings and locations, and Figure 21 illustrates interpolation of sparse data sets using different techniques.

2.6 Summary

This chapter covers the relevant differences between directional and omnidirectional antennae, as well as necessary concepts and terminology for radiolocation. It also discusses radiolocation applications, including roles such as emergency response, and offensive roles such as wardriving and CNO. Finally, sparse data sets and interpolation techniques are discussed.

III. Prototype Design

3.1 Overview

This research presents data and analysis of radiolocation using a directional antenna and techniques described in Chapter II. This chapter describes the design and construction of both hardware and software of the prototype. The prototype is designed with two primary goals in mind:

- **Low Cost.** A significant consideration for this research is the potential for low-cost applications. Prototype hardware is limited to commercial commodity hardware, and selection of the software framework and any libraries imported into the software project is limited to free OSS.

- **Low Weight.** This research intends to explore ways to quickly and accurately locate distant Wi-Fi access points from a UAV platform. Prototype hardware is selected that mimics the capabilities of a drone platform, namely low weight and antenna rotation control. Maximum prototype payload capacity is limited to 500 g, a reasonable payload for medium to large consumer UAVs [24].

Section 3.2 provides an overview of each hardware component with detailed technical specifications. Section 3.3 reviews the software implementation of the prototype and includes a detailed review of each component of the `localizer` framework.

3.2 Prototype Hardware

The hardware is limited to commercially-available, inexpensive products that are light enough to be carried by a consumer-grade UAV. The only prototype hardware expected to be migrated to a UAV prototype is the computer, computer power source, and antenna. The other components have roles that are filled by the UAV navigation computer and the UAV itself.

The hardware is reviewed in detail here and summarized in Tables 1 and 2. A schematic of the hardware is shown in Figure 10.

Computer. A Raspberry Pi 3 Model B is an ideal prototype computer for this application. Its selling price of $35 and 45 g weight fit within the project goals of low cost and weight. The Pi's 4-core architecture, the ready availability of Linux distributions and pre-compiled applications, and abundant general purpose input and output (GPIO) pins for controlling additional hardware make it a flexible and capable platform.

Computer Power Source. The Raspberry Pi draws a maximum current of 712 mA during the experiments detailed in Chapter IV. The selected $20 Letv LeUPB-211D Super Power Bank provides nearly 9.46 Ah (at a maximum observed voltage of 5.1 V) and is sufficient to power the Raspberry Pi for 13.3 h ($\frac{9.46\,\mathrm{Ah}}{712\,\mathrm{mA}}$) and longer than any current commercial drone can remain airborne. At 276 g, this is the heaviest UAV component in the prototype. A smaller battery with, for example, half the capacity would be sufficient, and significant weight may be saved by selecting one that is matched to the flight time of the device. This power source may be eliminated entirely if the UAV power supply provides capabilities to power other devices; however, flight time would be reduced slightly due to the increased power load from the Raspberry Pi.

Antenna. Many variations of directional antennae exist, nevertheless, the best candidate for this research that is adequately small, light, inexpensive, and commercially-available at the time of research is the Danets USB-Yagi TurboTenna yagi antenna. It weighs 137 g and measures 31.5 cm long, making it light and small enough to be carried by consumer-grade UAVs. It presents a cross section that is relatively small, ideal for a UAV in a windy environment and is visible in Figures 11 and 12.

Signal Receiver. The DNX10NH-HP USB Wi-Fi network interface reliably enters monitor mode and captures packets and compares well to the popular Alfa AWUS036H USB wireless adapter commonly used for wireless CNO. The DNX10NH-HP entered monitor mode and captured packets with zero loss throughout all the experiments. Its 35 g weight keeps the total weight within the desired maximum of 500 g.

Global Positioning System Module. The GPS module GlobalSat BU-353S4 is inexpensive and representative of a cheap, light, commercial GPS receiver. This module contains a SiRF Star IV GPS chipset that provides positional accuracy of less than 2.5 m. A GPS module may not be necessary on a UAV prototype, which usually have on-board GPS.

Antenna Motor. A motor is necessary to simulate the antenna rotation that a UAV platform may easily accomplish by its airborne mobility, or by being equipped with a gimbal that could rotate. To accurately determine the bearing at each phase of the capture, a stepper motor is used. The motor selected is a 2.0 A bipolar motor with a 1.8° step angle that provides 0.59 Nm of torque. This motor performed accurately and reliably during all experiments, rotating thousands of times without losing steps or becoming disoriented.

Motor Driver. A stepper motor driver is necessary to ensure smooth motor movement using microstepping; a purpose-built driver simplifies the necessary code to achieve microstepping and smooths antenna rotation. The motor controller selected is the MYSWEETY TB6600 Stepper Motor Driver, reportedly capable of driving 4 A from 9 V to 42 V. With microstepping the motor can rotate with a precision of 6400 steps per rotation.

To drive the motor, the Raspberry Pi simply sends a signal from its GPIO pins to the pulse (`PUL`) driver input. The Raspberry Pi does not have a real-time operating system; if the motor driver is controlled directly by GPIO, the signaling may be interrupted by operating system preemption. To ensure that this pulse signal is not interrupted or preempted by the Raspberry Pi operating system, hardware-timed pulse width modulation (PWM) signals are used to ensure that even when the Raspberry Pi's processor is under heavy load, the motor movement remains smooth.

Motor Power Source. Due to the length of the captures and the power required to drive the stepper motor (peak 2 A at 13 V), the author's truck is an acceptable choice for the motor power source. The truck's alternator provided more than 13 V and a stable platform for the prototype, which also benefited from the unobstructed position that the elevation provided. Figure 11 shows the prototype fixed atop the vehicle during data capture.

Miscellaneous Components. Other hardware used to build the prototype include MakerBeam t-slot aluminum extrusions for the frame, a breadboard, a ribbon cable, a GPIO breakout shield to simplify stepper motor control, and a rectangular Plexiglas piece as the prototybe base. Other items used in the construction of the prototype include a NEMA 17 mounting bracket and a 5 mm coupler.

Hardware Cost and Weight. An airborne prototype does not require all of the hardware listed in Table 1; for example, a motor for rotating the antenna is unnecessary based on an assumed quadrocopter or similar design that is capable of rotation. The components that are required for an airborne prototype are listed in Table 2 with their respective cost and weight. As shown, the cost of hardware is low, and the cumulative weight is less than the proposed maximum of 500 g.

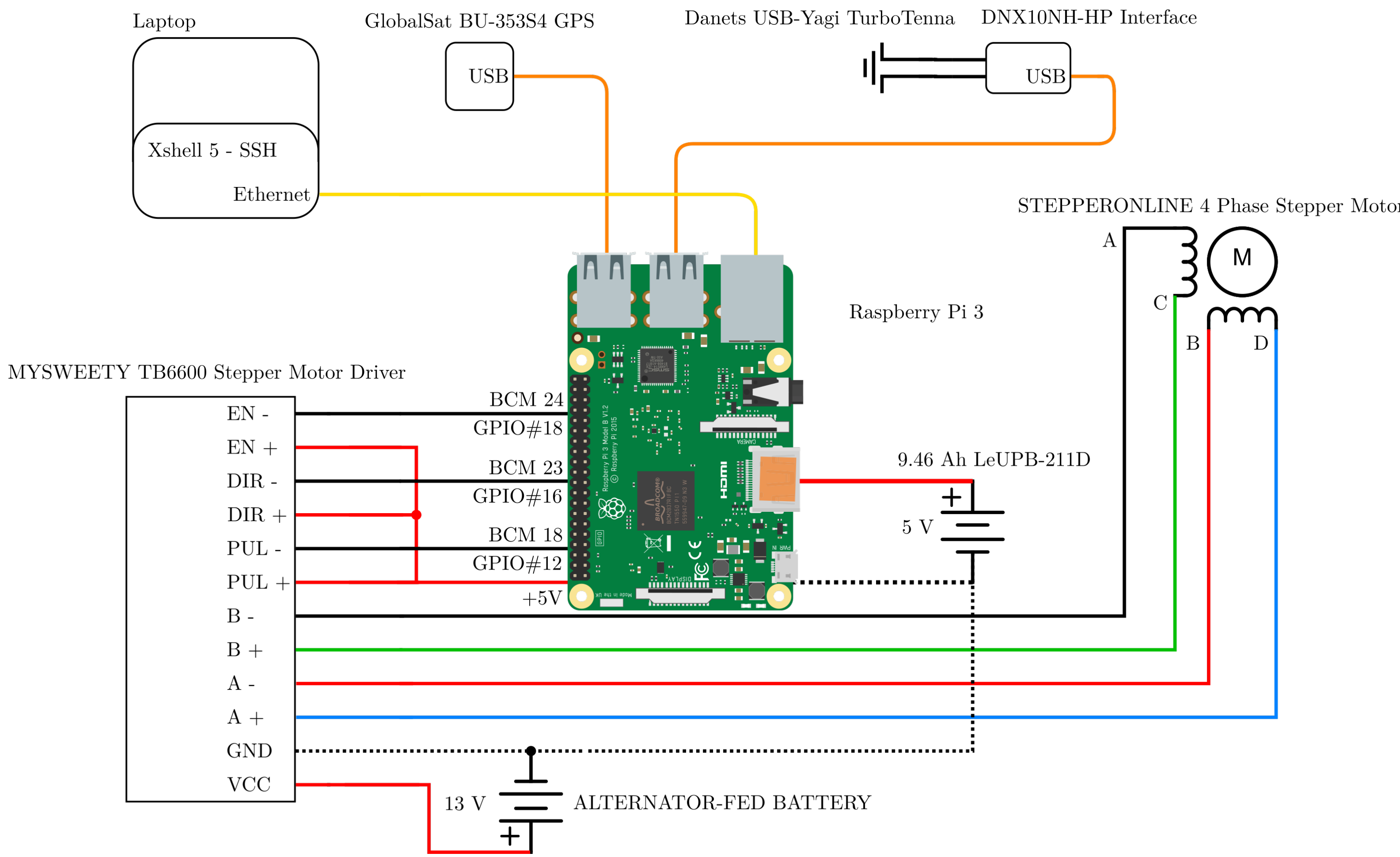


Figure 10. Prototype Schematic

Table 1. Prototype Hardware Overview

Item	*Model / Version*
Processor	Raspberry Pi 3 Model B V1.2 - Raspbian Stretch (4.9)
– Power	Letv LeUPB-211D 13.4 Ah (3.64 V)
Antenna	Danets USB-Yagi TurboTenna
Receiver	USB WiFi Interface - DNX10NH-HP
GPS Module	GlobalSat BU-353S4
Motor	NEMA 17 2.0 A Bipolar Stepper Motor
Motor Driver	MySweety Microstep Stepper Driver - TB6600
– Power	Alternator-Fed Battery (Ranger)
Structure	MakerBeam 10mm Aluminum Extrusions
Motor Mount	NEMA 17 Steel L Bracket
Motor/Antenna Coupler	Aluminum Flex Shaft 5mm to 5mm coupler
Antenna Mast	5mm Steel Bolt
GPIO Components	Standard Breadboard, GPIO Breakout, Ribbon Cable

Table 2. Airborne Prototype Hardware Cost and Weight

Item	*Price*	*Weight*
Processor	$35	45 g
– Power	$20	276 g
Antenna	$113	137 g
Receiver	$0 (incl. with antenna)	35 g
Totals	*$168*	493 g

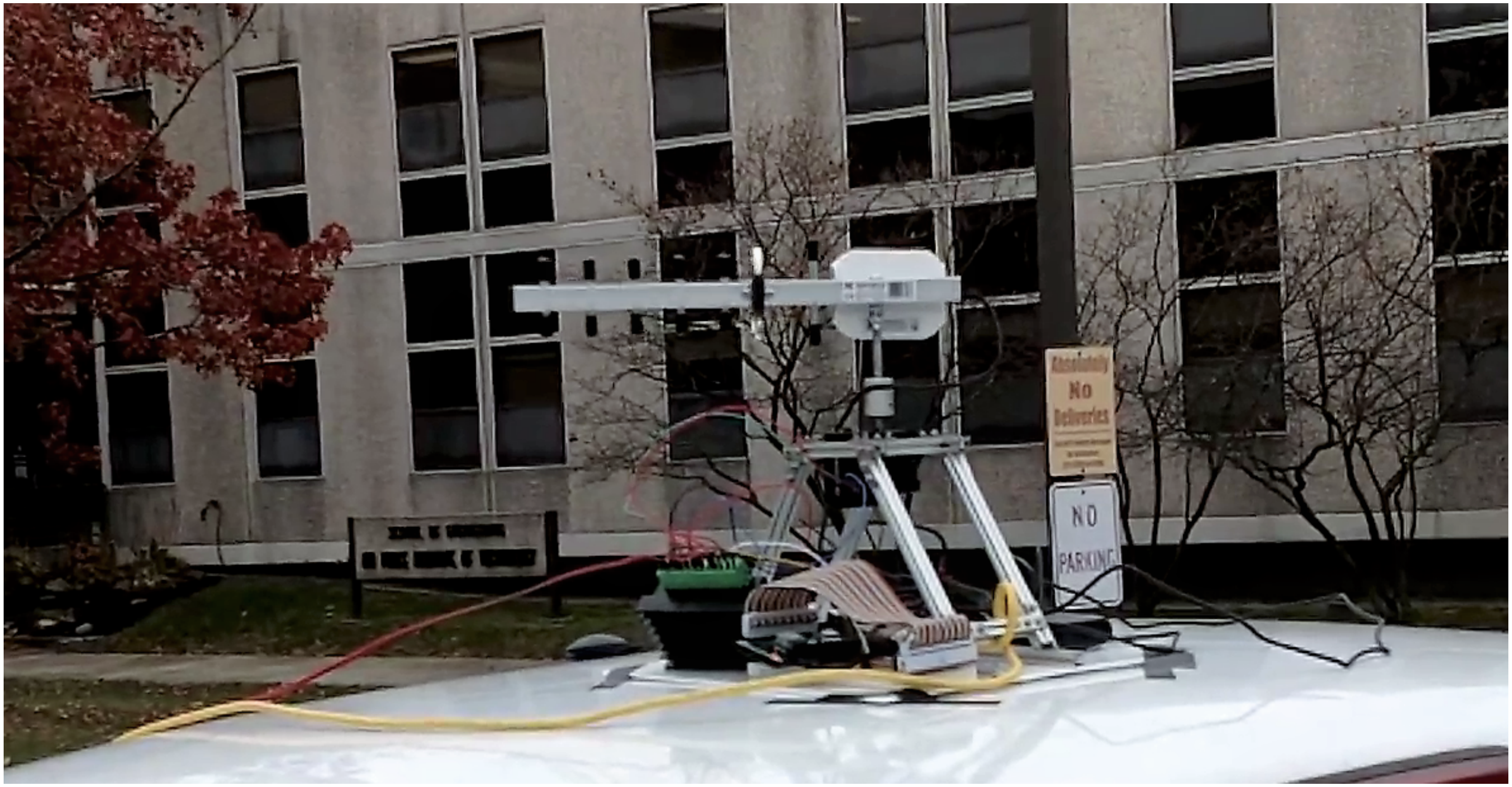


Figure 11. Experiment Platform and Power Source

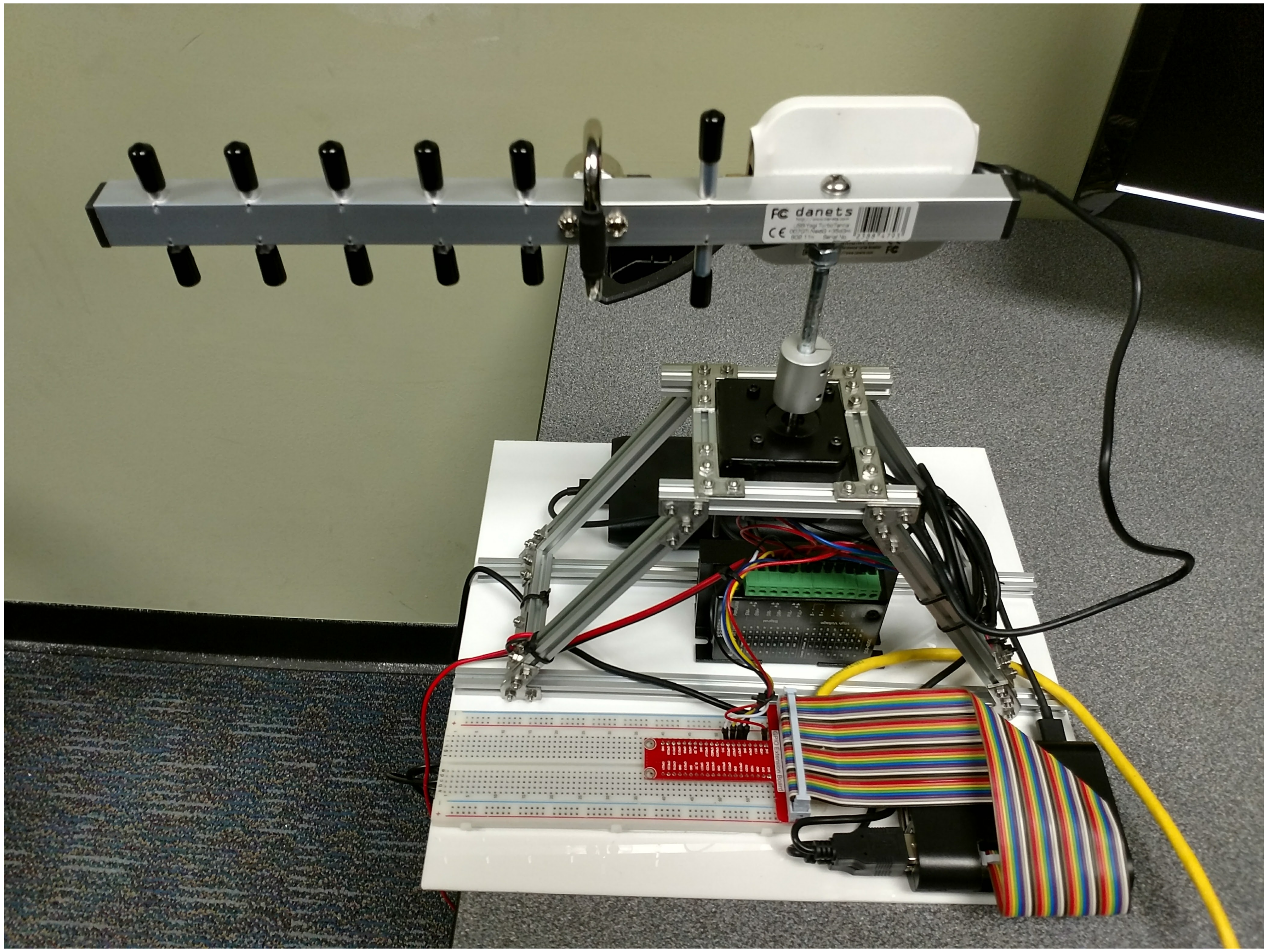


Figure 12. Assembled Prototype

3.3 Prototype Software

This research requires specialized software to manage and synchronize the different prototype components. The resulting software is `localizer`, a framework written in Python 3.5.3 and composed of modules organized by their respective capture functions.

- **Interactive Shell.** A command line interface is the central component of `localizer`. The interactive shell allows a user to set parameters for capture, such as degrees of rotation, rotation speed, channel hop rate, and other parameters for managing a live capture. Once configured, the user may initiate the capture, which sets in motion the orchestrated efforts of several threads to synchronously capture data at the specified parameters. After the capture, the shell displays its best guess as to the bearing of all detected BSSIDs. The user has the option to conduct a focused capture, which focuses the sweep on a particular range and a channel specific to the targeted BSSID.

- **Batch Capture.** The shell also contains a subroutine for batch processing that reads in capture configuration files. These may have a variable number of captures defined within, as well as a passes number set. The passes number signifies how many times each capture should be repeated. Batch capture is the primary mode for capturing the data that is analyzed in this research.

- **Batch Processing.** Finally, `localizer` has a batch processing command-line feature that walks a designated directory and all subdirectories, searching for unprocessed data sets. If found, it uses multiprocessing to process the data for later analysis.

`localizer` uses Python 3.5.3, chosen for its flexibility and cross-platform characteristics, as well as the ease of multithreading and multiprocessing. Multithreading is used in the capture process to synchronize many asynchronous processes, while multiprocessing is used during processing of the captured data. The project dependencies used in `localizer` are listed in Table 3 with their respective version number and the function they provide to the framework.

Table 3. `localizer` **Dependencies**

Package	*Version*	*Function*
`gpsd`	3.16-4	Provide GPS data from GPS module
`gpspipe`	3.16-4	Log GPS data to disk
`iwconfig`	30	Get Wi-Fi adapter settings, set monitor mode/channel
`iwlist`	30	Get Wi-Fi interface current channel
`ifconfig`	2.10-alpha	Prepare Wi-Fi interface for monitor mode
`dumpcap`	2.2.6	Capture packets from Wi-Fi interface
`pigpio`	64	Provide reliable hardware PWM to stepper motor

3.4 Modules

The `localizer` project is organized into different modules based on their unique role in the data acquisition process. The `localizer` manual is provided in Appendix A and the complete project source code is given in Appendix B. This section reviews the most significant modules and includes references to the relevant source code.

3.4.1 `shell.py`.

Reference Appendix B.2.3 for `localizer/shell.py` source code.

This module provides the first two roles, namely interactive capture and batch capture by making extensive use of subclassing the Python `Cmd` class. As demonstrated in the manual, many commands are available to set up capture parameters, view the current state, and execute a capture. The batch capture mode enables `*-capture.conf` file import and batch capture.

3.4.2 `capture.py`.

Reference Appendix B.3.1 for `localizer/capture.py` source code.

This module performs the bulk of all capture thread synchronization. Each thread and its respective role in the data capture process is detailed in Table 4, whereas Table 5 lists the capture output of each thread.

The function `capture` manages each thread in the following sequence:

1. **Initialize Environment.** Set up capture paths and filenames based on provided parameters. Create the event flags used later for thread synchronization

2. **Set Up Threads.** Initialize the four required capture threads, CaptureThread, GPSThread, ChannelThread, and AntennaThread. Start each thread to give each time to perform initialization functions. Each thread awaits an event flag to signal when it should begin its capture routine.

3. **Wait for Precise GPS Fix.** Poll the GPS provider until a precise location is indicated.

4. **Wait for CaptureThread.** CaptureThread raises a synchronization flag to indicate that it has successfully started capturing packets. All other threads are waiting on this flag, and start their respective captures once it is raised.

Table 4. Capture Thread Roles

Thread	*File*	*Role*
CaptureThread	`capture.py`	Start `dumpcap`, wait for feedback, trigger other threads
GPSThread	`gps.py`	Capture GPS NMEA data for the capture duration
ChannelThread	`interface.py`	Change monitored channel at a given hop rate
AntennaThread	`antenna.py`	Rotate antenna at a given rate and degrees

Table 5. Capture Thread Outputs

Thread	*Capture Output*
CaptureThread	capture start & stop times, <timestamp>.pcapng
GPSThread	average coordinate, <timestamp>.nmea, <timestamp>-gps.csv
ChannelHopper	none
AntennaThread	capture start & stop times

5. **Wait for Other Threads.** Wait for the specified capture duration while all threads perform their respective function as described in Table 4. The capture duration is provided interactively by an operator or by a batch capture script.

6. **Collect Results.** After the capture duration has elapsed, collect the results from each thread from its respective queue. The output from each thread is detailed in Table 5.

7. **Write Metadata.** Write the capture details to `<timestamp>-capture.csv`. This file is comprised of important meta data as described in Table 6 and is used during capture processing.

8. **Optional: Predict WAP Bearings.** If directed, `capture` processes the collected data by grouping beacons from the same BSSID into discrete series. Each series is interpolated using an optimal interpolation method, and a prediction is made for the bearing to the emitter. If this happens during an interactive capture, the enumerated BSSID are displayed and the operator may select one

Table 6. Metadata Fields

Field	*Type*	*Description*
`name`	string	The capture name; if none is given, timestamp of the capture
`pass`	int	The capture pass number
`path`	string	The path where capture data is recorded
`pcap`	string	The file name of the packet capture
`nmea`	string	The file name of the raw NMEA capture
`coords`	string	The file name of the logged GPS coordinates
`iface`	string	The capture interface
`duration`	int	The number of seconds to capture data
`hop_int`	double	The interval in seconds between channel hops
`pos_lat`	double	The mean latitude of the capture
`pos_lon`	double	The mean longitude of the capture
`start`	double	The timestamp of when the capture began
`end`	double	The timestamp of when the capture concluded
`degrees`	int	The number of degrees over which the capture is conducted
`bearing`	int	The initial bearing of the capture
`focused`	string	The BSSID of the targeted WAP if capture is focused

to target with a *focused* capture by issuing the command `> capture` followed by the specified BSSID number. Batch captures may be configured to automatically perform focused captures for each detected BSSID or specified white-listed BSSIDs.

9. **Clean up Environment.** Allow each thread to clean up and join the main thread.

CaptureThread. This class extends the Python `threading.Thread` class and spawns an instance of `dumpcap`, a part of the `tshark` package which is used in Wireshark packet capturing. `dumpcap` is used because of its low resource requirements. Because `dumpcap` is relatively slow to start capturing packets, the other capture threads wait for a flag from CaptureThread indicating that `dumpcap` has successfully begun capturing packets.

3.4.3 `gps.py`.

Reference Appendix B.3.3 for `localizer/gps.py` source code.

This module exclusively deals with GPS initialization and capture.

GPSThread. When triggered by CaptureThread, this thread performs two functions:

- **Poll for GPS Data.** The thread manually polls the system GPS provider (`gpsd`) for GPS data every 1 s, the maximum rate that the prototype GPS module supplies updated GPS readings. The results are written to the file `<timestamp>-gps.csv`.
- **Spawn `gpspipe`.** In addition to polling `gpsd`, the thread uses `gpspipe` to pipe raw NMEA GPS data from `gpsd` to a file ending in `<timestamp>.nmea`.

3.4.4 `interface.py`.

Reference Appendix B.3.4 for `localizer/interface.py` source code.

The `wifi` module manages the Wi-Fi radio, including entering and exiting monitor mode, getting wireless adapter information, and setting the interface channel.

ChannelThread. The `wifi` module has the ChannelThread class, which, when triggered by CaptureThread, manages cycling through channels during a wide capture, or holding the channel steady during a focused capture. The interval between channel hops, as well as the hop pattern, is configurable, and optimal values are discussed in Section 5.3.

3.4.5 `antenna.py`.

Reference Appendix B.3.2 for `localizer/antenna.py` source code.

This module has the important duty of managing the stepper motor, and by extension, antenna bearing. `antenna.py` starts with global variables and initialization code that ensures the right system programs are available, such as `pigpiod`, the Python and Raspberry Pi library that provides hardware-based PWM pulses for the stepper motor.

AntennaThread. Most antenna functionality is encapsulated in AntennaThread. When initialized, this class resets the antenna to a specified bearing. When triggered by CaptureThread, AntennaThread rotates the antenna to a given bearing at a specified rotation rate. Optionally, if provided a reset bearing, the thread resets the antenna once primary rotation is complete.

Because of the wire from the collector (Wi-Fi adapter) to the processor (Raspberry Pi), this class has the responsibility to rotate the antenna given arbitrary rotation angles while ensuring that the antenna does not rotate too far in either direction. The class function `determine_best_path` takes a new bearing and the proposed degrees of travel and determines the ideal path. If possible, this function returns the shortest path to the new bearing while ensuring that the rotation avoids tangling the interface cable (e.g., rotating too far in the same direction causes the cable to bind up the antenna and miss steps or stop rotating entirely). A UAV prototype does not suffer from this limitation, unless a gimbal is used, because the antenna does not rotate independently of the UAV.

Once an ideal travel path has been determined, the `rotate` function generates pulse waves to be provided to `pigpiod`, which translates them into PWM pulses that drive the motor at the desired rate and to the desired distance. Acceleration and deceleration of the antenna when it starts and stops is included in each wave to help ensure accurate antenna rotation.

Reset Rate. When resetting the antenna, a special antenna reset rate is used. When resetting over long distances, a high speed is appropriate, however rotating the prototype at short distances and high speeds causes the stepper motor to miss steps, invalidating all data that is captured afterward. To compensate for this, a smoothing function based on a symmetric sigmoid is used to ensure no missed steps would occur. The sigmoid reset rate problem is discussed further in Section 5.2.2.

3.4.6 `process.py`.

Reference Appendix B.3.5 for `localizer/process.py` source code.

Capture data in the form of metadata, GPS positions, and packet captures must be processed in order for it be readily analyzed or to facilitate a prediction as to the bearing of detected emitters.

Except for several helper utilities, the `process.py` module has only two primary functions:

- `process_capture`. This function accepts a path to a capture meta file, as described in Section 3.4.2 and detailed in Table 6. The meta file is ingested, along with the capture files described in Table 5. Each packet that is captured is parsed for important information such as BSSID, SSID, RSSI, and WAP security details. Antenna bearing is derived from the packet timestamp and the data provided by the AntennaThread timing results. The processed results are tabulated and written to disk as a `<timestamp>-results.csv` file for future analysis and optionally used directly to predict the bearings of any BSSIDs detected during the capture.

- `process_directory`. This function walks through a given directory and all of children directories identifying and collecting unprocessed capture sets. It sends the discovered unprocessed capture sets to a multiprocessing pool that processes each capture in parallel.

3.4.7 `locate.py`.

Reference Appendix B.2.2 for `localizer/locate.py` source code.

This small module provides an important function to the `localizer` project of data interpolation. Interpolation, which is discussed further in Chapter V, can reduce the localizing error by over 30° in some cases. In short, this module takes sparse data sets of beacon intensity as a function of bearing and fills the missing data using a variety of techniques. The most effective interpolation methods are discussed in Chapter V.

3.5 Summary

The hardware described in this chapter meets the requirements of low weight and cost, while the software meets the requirements that it be open source and capable of performing the functions necessary to gather the data as outlined in Section 3.4.2.

IV. Methodology

4.1 Overview

This research proposes a unique method of locating Wi-Fi emitters from an UAV using a directional antenna. This chapter describes the experiment environment and identifies the metrics and parameters necessary to measure the performance of the prototype. This chapter covers experimental treatments that deliver data for analysis and parameter discovery.

4.2 System Under Test

Figure 13 displays the System Under Test (SUT) and Component Under Test (CUT) diagrams. The workload factors consist of wide capture parameters and focused capture parameters, described in Section 4.4. The system parameters, comprised of computing and prototype parameters (covered in Section 3.2) and constant parameters (covered in Section 4.4 and Table 8) are held constant throughout all experiments. The system metrics are detailed in Section 4.5 and shown in Table 9.

4.3 Experiment Objectives

To test the research hypothesis as discussed in Section 1.4, this research seeks to discover optimal values for the parameters that are identified in Section 4.4 and that maximize the performance of the proposed localization method. These optimal parameters are discovered through multiple experiments described in this chapter and analysis covered in Chapter V.

This research also seeks to discover whether accurate WAP coordinates may be discovered by repeating the wide capture process from multiple locations. In pursuit of this end, data is gathered from multiple positions.

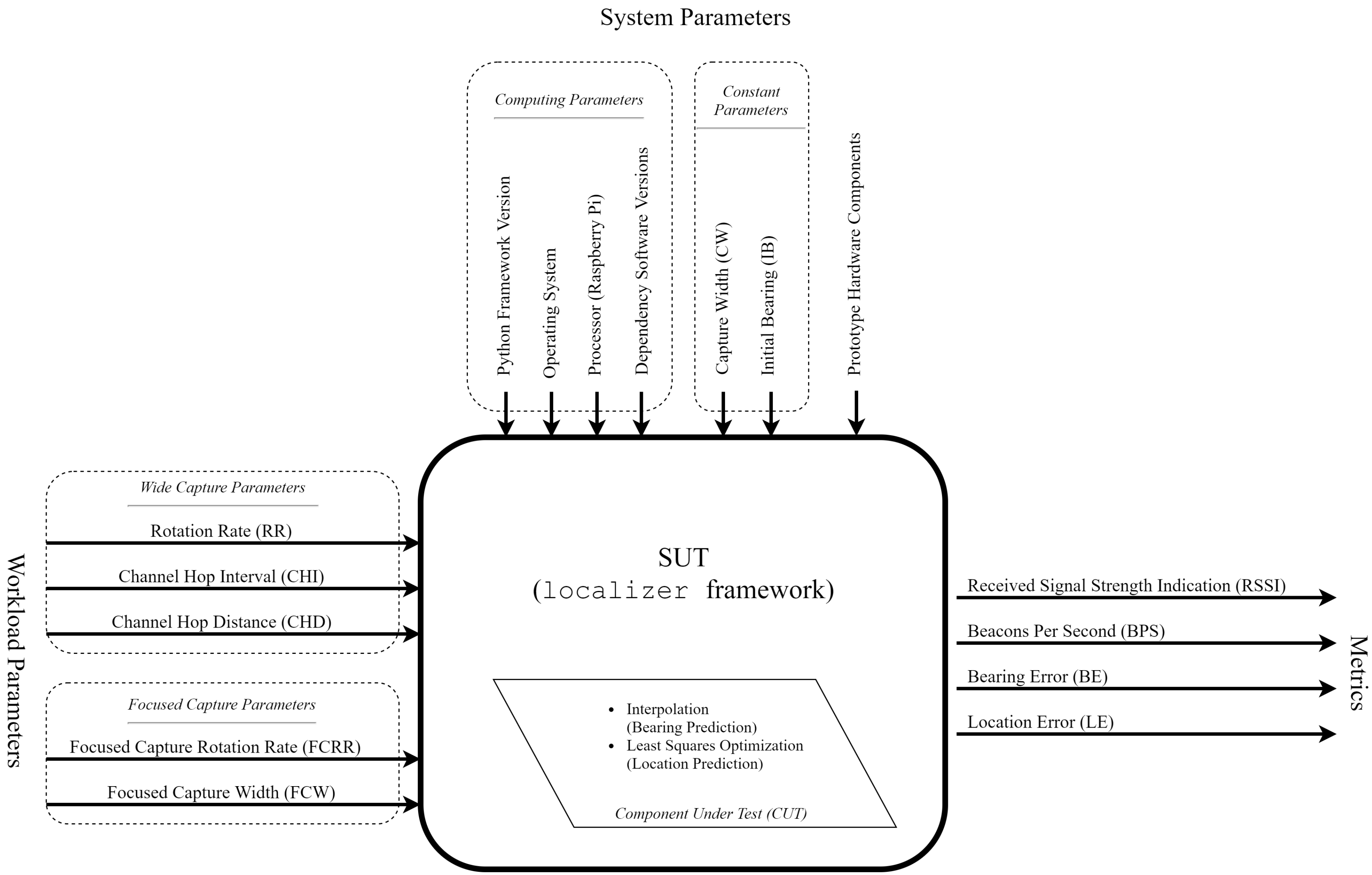


Figure 13. System Under Test and Component Under Test

4.4 Parameters

The parameters identified in Figure 13 are outlined in this section and summarized in Table 7.

1. **Rotation Rate (RR).** Because most captures are assumed to be a single rotation, and because captures faster than one revolution per second are impractical, this parameter is displayed as seconds per revolution instead of its inverse.

   These treatments prioritized gathering data for analysis and measuring bearing error (BE) and location error (LE). For this purpose, rotation rate (RR) is optimized to find the highest beacons per second (BPS).

2. **Focused Capture Rotation Rate (FCRR).** Like RR, focused capture rotation rate (FCRR) is the rotation rate for a focused capture where the channel is fixed. This thesis hypothesizes that keeping the channel fixed during a capture reduces missed beacons, increasing observed beacons and enabling accurate high-speed captures.

3. **Channel Hop Interval (CHI).** Channel hop interval (CHI) is the amount of time in time units (TUs) that the receiving Wi-Fi adapter waits on a channel in monitor mode before moving on to the next channel.

   In an attempt to avoid issues with certain WAPs in that a chosen CHI "misses" the WAP's beacon (or some subset of them) because of unintended synchronization with the WAP beacon emissions, CHI values are defined by the following set notation:

$$\{x \,\epsilon\, \mathbb{Z}_{\geq 100} \mid gcd(x, 100) = 1\} \tag{2}$$

where x is an element of the integer set ($\mathbb{Z}$) greater than 100 and where the greatest common denominator (*gcd*) of x and 100 is 1.

In other words, the hop interval may only be a relatively prime number of TUs that is at least 100 TU, the standard beacon emission rate. Appendix C.2 contains the script used to generate possible coprime hop intervals.

4. **Channel Hop Distance (CHD).** Channel hop distance (CHD) is the number of channels to move when hopping. For example, consider the standard 802.11b and 802.11g channels in the United States of 1-11. Starting at 1, a channel hop distance of 1 would step through each channel sequentially. A channel hop distance of 2 would step through every other channel in the sequence of 1,3,5, and so on.

   The channels are close enough that it is common for traffic transmitted on a particular channel to be observable on adjacent channels. An optimal CHD could potentially increase the number of beacons, improving bearing and location predictions.

5. **Focused Capture Width (FCW).** Conducting an optimal focused capture requires a focused capture width (FCW) that provides the ideal trade off between BE and detection time (DT).

Two parameters that are held constant for the duration of the experiment are capture width (CW) and the initial bearing (IB), representing the amount of antenna rotation for a wide capture and the starting bearing, respectively. The held-constant value of CW represents the assumption that a wide scan always rotates 360° to locate WAPs in every direction, and the held-constant value of *initial bearing* made running experiments consistent and verifiable throughout by comparing the actual antenna bearing with the reported antenna bearing. In practice, before each capture, the antenna is directed to 0°, magnetic North. At the end of the capture, the antenna bearing is compared to this initial value. If it differed from 0° the capture is invalidated. The `localizer` framework is verified to maintain an accurate bearing after more than 24 hours of continuous, randomized captures. The parameters CW and IB are summarized in Table 8.

4.5 Metrics

The goal of this research is to determine WAP bearings and locations. With this goal in mind, the performance of the system can be measured directly by the error in producing a bearing and a location. Ancillary metrics are listed first, followed by the two primary metrics which are summarized in Table 9.

1. **Received Signal Strength Indication (RSSI).** This metric is introduced in Section 2.3.1. RSSI is used heavily in this experiment to measure the received strength of the beacon.

2. **Milliwatt (mW).** In this research, milliwatt is a more useful measurement metric than RSSI for localizing emitters, as shown in Section 5.4.

3. **Beacons per Second (BPS).** All things equal, more beacons provide more data to use in localizing WAP emitters, and as shown in Chapter V, more beacons improved localization performance significantly. One of the major parameters in this experiment is capture duration, or speed of the antenna rotation. Beacons per second is an ideal metric to determine optimal rotation speed, and serves to identify optimal channel hopping rate and channel hopping distance.

   The beacons per second metric may be expressed as a positive integer, BPS, which is the ratio:

$$BPS = \frac{BO}{CD} \tag{3}$$

Table 7. Experiment Parameters

Parameter	*Units*	*Range*	*Proposed Values*
Rotation rate (RR)	$\frac{s}{rev}$	0 to ∞	$\{5, 10, 15, 20, 25, 30\}$
Focused capture rotation rate (FCRR)	$\frac{s}{rev}$	0 to ∞	$\{5, 6 \ldots 12, 13\}$
Channel hop interval (CHI)	TU	100 to ∞	$\{109, 119, 129 \ldots 199\}$
Channel hop distance (CHD)	ch	0 to ∞	$\{1, 2, 3, 4, 5\}$
Focused capture width (FCW)	°	0 to 360	5 - 360

Table 8. Held-Constant Parameters

Parameter	*Units*	*Held-Constant Value*
Capture width (CW)	degrees	360
Initial bearing (IB)	degrees	0

where beacons observed (BO) represents the number of beacons observed for a particular capture and capture duration (CD) represents the time spent conducting the capture, rotating the antenna and capturing beacons.

BPS_w and BPS_f. In reality, there are two measurements for BPS - one for wide captures and another for focused captures. BPS may then be designated as BPS_w or BPS_f respectively. BPS without subscript may be assumed to be used for wide captures with channel hopping, BPS_w. BPS is measured in beacons per second ($\frac{b}{s}$).

The standard beacon rate is ten beacons per time unit. In wide capture mode, the prototype only monitors a single channel at a time and each WAP is on a distinct channel. Ignoring cross-channel observations (where traffic from one channel is observed by an adapter monitoring another), the upper limit for BPS is therefore one beacon per 100 TU ($\approx 9.77\,\frac{b}{s}$). Factoring the directionality of the antenna at a very generous 120° beam width, the maximum expected BPS is reduced to 1 every 300 TU ($\approx 3.26\,\frac{\mathrm{b}}{\mathrm{s}}$). It is possible the observed BPS is higher due to reflections reducing the directionality reduction. The lower limit of BPS is set to the worst case of $0\,\frac{\mathrm{b}}{\mathrm{s}}$.

In the case of focused capture modes, $\mathrm{BPS_f}$ is expected to be higher than $\mathrm{BPS_w}$ because focused captures do not incur any channel switching penalty. This metric is expected to be 50% higher than $\mathrm{BPS_w}$, which is $4.89\,\frac{\mathrm{b}}{\mathrm{s}}$.

4. **Capture Overhead (CO).** This metric is the amount of time overhead necessary to conduct a capture. It is the difference of DT, which is the time that `localizer` is busy conducting the capture and CD, the time spent actively capturing:

$$CO = DT - CD \tag{4}$$

Capture overhead (CO) is a function of system performance and when running on the same hardware (Raspberry Pi), is assumed to remain constant regardless of the number of beacons observed and the capture duration. This metric may be used for both wide and focused captures since the system processes are the same.

5. **Capture Processing Overhead (CPO).** Capture processing overhead (CPO) is the amount of time necessary to process a data set and generate predicted bearings for any observed WAPs. This metric is a function of the number w of WAPs observed:

$$CPO(w) = aw + b \tag{5}$$

where a and b constants based on processor hardware.

Table 9. Performance Metrics

Metric	*Units*	*Accepted Range*	*Expected Range*
Received signal strength indication (RSSI)	dBm	$-\infty$ to ∞	$-80\,\mathrm{dBm} < RSSI < -30\,\mathrm{dBm}$
Milliwatt (mW)	mW	0 to ∞	$0\,\mathrm{mW} < mW < 1\,\mathrm{mW}$
Beacons per second (BPS) - Wide	$\frac{\mathrm{b}}{\mathrm{s}}$	0 to ∞	$0\,\frac{\mathrm{b}}{\mathrm{s}} < BPS_w < 3.26\,\frac{\mathrm{b}}{\mathrm{s}}$
Beacons per second (BPS) - Focused	$\frac{\mathrm{b}}{\mathrm{s}}$	0 to ∞	$0\,\frac{\mathrm{b}}{\mathrm{s}} < BPS_f < 4.89\,\frac{\mathrm{b}}{\mathrm{s}}$
Capture overhead (CO)	s	0 to ∞	$0.5\,\mathrm{s} < CO < 1\,\mathrm{s}$
Capture processing overhead (CPO)	s	0 to ∞	$aw + b < CPO < cw + d$
Bearing error (BE)	$^\circ$	0 to 180	$5^\circ < BE < 45^\circ$
Location error (LE)	m	0 to ∞	$LE < 10\,\mathrm{m}$

6. **Bearing Error (BE).** Establishing and maintaining a wireless connection with a target WAP is the primary role of the DWAP. This metric measures the difference between the bearing predicted by a localization attempt and the *true* bearing. This research is conducted under the bearing consistency assumption listed in Section 1.6. The impact of the possible difference between the true bearing and the bearing of strongest RSSI is supposed to be minimal, yet should be understood that this metric is measuring the error between what the prototype predicted as the bearing of strongest RSSI and the true bearing to the responsible WAP.

   The bearing accuracy metric may be expressed as a positive real number, BE, which is the difference of true bearing (TB) and peak RSSI (PR):

$$BE = |TB - PR| \tag{6}$$

7. **Location Error (LE).** The locations of Wi-Fi emitters such as WAPs and smartphones may be discovered given enough radiolocation data. The knowledge of these locations is valuable to wardroning UAVs, search and rescue UAV, and DWAPs to enumerate just a handful of the many other possible uses. This research seeks to derive location data for the experiment WAPs using the directional data produced by multiple capture sets.

   The location accuracy may be expressed as a positive real number, LE, which is the haversine (i.e., great-circle [25]) distance between the predicted position (PP) and true position (TP), where each position is comprised of two components latitude and longitude, (φ_1, λ_1) and (φ_2, λ_2) respectively:

$$LE = 2r \arcsin \sqrt{\operatorname{hav}(\varphi_2 - \varphi_1) + \cos(\varphi_1)\cos(\varphi_2)\operatorname{hav}(\lambda_2 - \lambda_1)} \tag{7}$$

   where

$$\operatorname{hav}(\Theta) = \sin^2\left(\frac{\Theta}{2}\right) \tag{8}$$

Variable r is the radius of the earth at a particular latitude φ, given by

$$r(\varphi) = \sqrt{\frac{(a^2 \cos\varphi)^2 + (b^2 \sin\varphi)^2}{(a\cos\varphi)^2 + (b\sin\varphi)^2}} \tag{9}$$

and where a and b are the equatorial radius (6 378 137 m) [26] and polar radius (6 356 752 m) [27] respectively.

4.6 Experiment Environment

All experiments are conducted at the AFIT campus at three locations, listed in Table 12, displayed in Figure 15, and centered around coordinates 39.782 755 6°, −84.083 002 8°.

WAPs. Ten WAPs are placed surrounding the capture locations and shown in Figure 14. The WAPs are configured with unique SSIDs and channels as recorded in Table 10, as well as their respective coordinates and unique BSSIDs. Table 11 shows the hardware and software version of each WAP.

Note: WAP 0 was installed in a public area, and between treatments 4 and 5 went missing. It was replaced by WAP 10, which is indicated by gray text in Tables 10 and 11. WAP 10 is configured identically to WAP 0 except for SSID and BSSID; a change in these values has no effect on these research experiments.

Each WAP is prepared by following the same procedure:

1. **Firmware Updated.** If an updated firmware is available, it is applied to the WAP.

2. **Reset to Factory Defaults.** Each WAP is reset to its factory defaults.

3. **WPA PSK Enabled.** WPA PSK security protocols are enabled on each WAP.

Table 10. Wireless Access Point Configurations & Locations

WAP	*Chan*	*SSID*	*BSSID*	*Lat*	*Lon*
0	10	RESEARCH_MULLINS_0	1c:7e:e5:30:57:4e	39.78249	-84.0839
1	1	RESEARCH_MULLINS_1	00:18:e7:e9:04:59	39.78229	-84.0838
2	2	RESEARCH_MULLINS_2	00:18:e7:e9:07:f5	39.78240	-84.0831
3	3	RESEARCH_MULLINS_3	00:12:17:9f:79:b6	39.78250	-84.0828
4	4	RESEARCH_MULLINS_4	00:16:b6:58:f3:0d	39.78287	-84.0828
5	5	RESEARCH_MULLINS_5	60:38:e0:06:2d:9c	39.78322	-84.0827
6	6	RESEARCH_MULLINS_6	60:38:e0:06:3a:d8	39.78346	-84.0827
7	7	RESEARCH_MULLINS_7	60:38:e0:06:34:e8	39.78329	-84.0831
8	8	RESEARCH_MULLINS_8	60:38:e0:06:34:ac	39.78325	-84.0832
9	9	RESEARCH_MULLINS_9	60:38:e0:06:3a:f0	39.78342	-84.0838
10	10	RESEARCH_MULLINS_10	1c:7e:e5:30:54:3e	39.78249	-84.0839

Table 11. Wireless Access Point Models & Firmware

WAP	*Model*	*Firmware*
0	DIR-615, HW E3	5.1
1	DIR-615, HW E3	5.11
2	DIR-615, HW E3	DD-WRT v24-sp2
3	WRT55AG, v2	1.67
4	WRT55AG, v2	1.67
5	WRT1200AC v2	2.0.4.173345
6	WRT1200AC v2	2.0.4.173345
7	WRT1200AC v2	2.0.4.173345
8	WRT1200AC v2	2.0.4.173345
9	WRT1200AC v2	2.0.4.173345
10	DIR-615, HW E3	5.1

4. **Exclusive 802.11g Mode Enabled.** The 802.11g standard is enabled on each WAP, and all other standards are disabled.

5. **Unique Channel Set.** Each WAP is assigned a unique channel, shown in Table 10.

6. **Install WAP.** Take each WAP to its location and plug into a power source.

Environment. The experiment location is chosen because each WAP is sheltered from the elements in secure locations; this selection avoided having to set up and take down the WAPs between experiments. Furthermore, the environment presents a realistic, real-world Wi-Fi environment, including commercial buildings and structures with existing 2.4 GHz 802.11g equipment behind walls constructed of brick, concrete, steel, glass, and other materials. The AFIT campus is selected due to its suitability under these considerations.

Compared to the environment that a UAV would be conducting these actions, this ground-based experiment suffers a significant disadvantage due to electromagnetic issues. While emitters within structures mimics the types of targets a DWAP would target, the structures surrounding the capture location do not accurately simulate an airborne environment, primarily regarding reflections. Analysis of this effect is discussed in more depth in Chapter V, however, it is important to note here. Furthermore, the selected location did not test the distance limits of the directional antenna, which is not a key part of these experiments.

Table 12. Capture Locations

Capture Location	*Latitude*	*Longitude*
1	39.7827250	-084.0830556
2	39.7827417	-084.0828778
3	39.7828778	-084.0830639

Figure 14. Wireless Access Point Locations (Map data: Google)

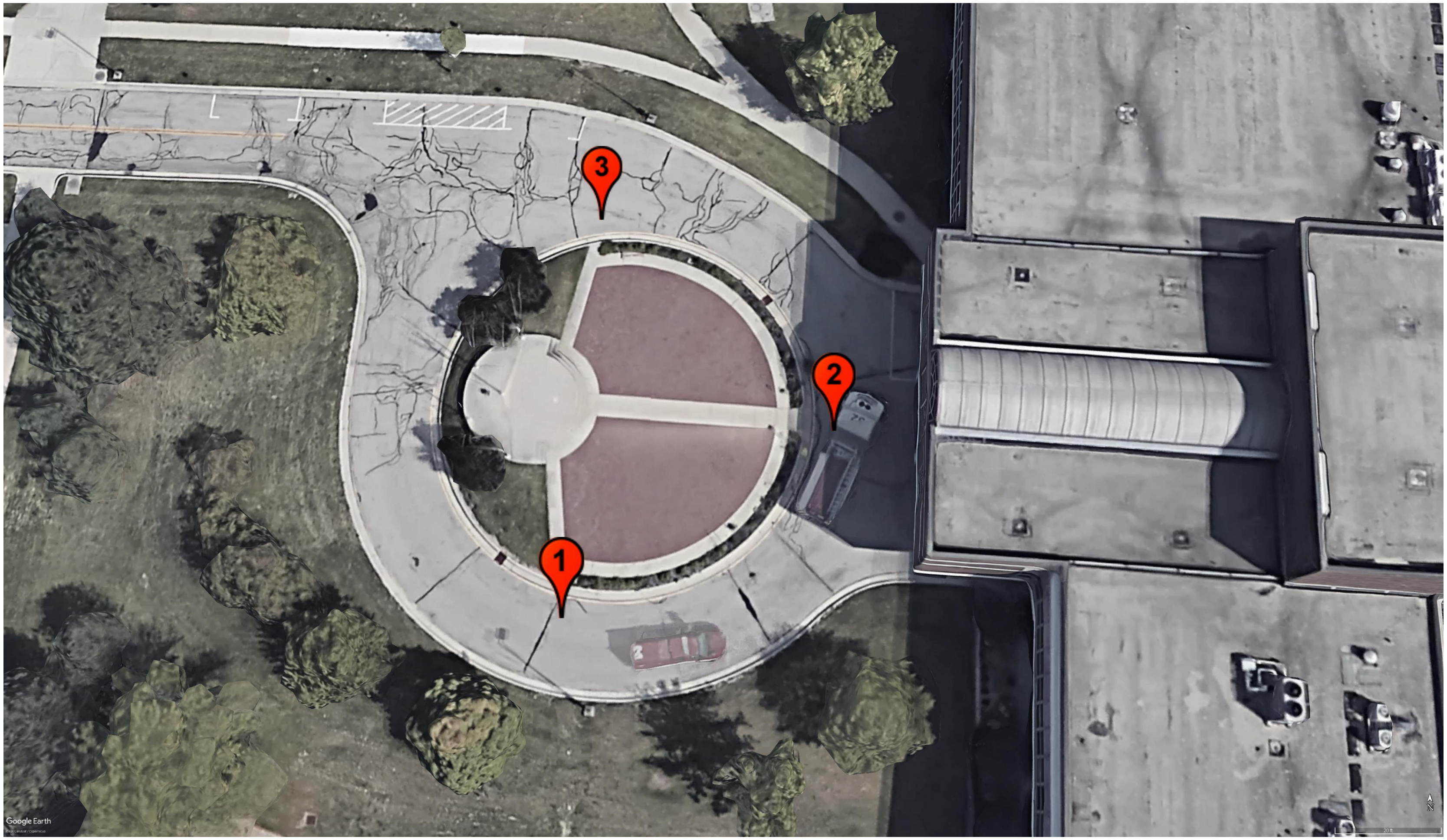


Figure 15. Capture Locations (Map data: Google)

4.7 Experimental Design

4.7.1 Treatments.

This experiment is conducted in three sets of treatments, each of which is detailed in this section. The following treatments are analyzed in like order starting in Section 5.3, and summarized in Tables 13, 14, and 15. The capture configuration files used in executing these treatments in `localizer` are listed in Appendix D.

Parameter Discovery. The first goal of this experiment is to identify the parameter values (Table 7) that produce optimal metrics (Table 9).

A treatment is prepared for each of the parameters to capture data under each of the proposed values, with other parameters held constant to a reasonable value. All parameter discovery treatments are conducted at capture location 1 as listed in Table 12 unless otherwise noted.

1. *Rotation rate* This parameter optimization has two treatments:

   (a) $RR = \{5, 10, 15, 30\}$

   Passes: 30

   Held-constant parameter: $CHI = 130\,\text{TU}$

   Held-constant parameter: $CHD = 1$

   Note: This treatment uses a value of CHI that is not relatively prime with the default beacon rate of 100 TU; this treatment was performed before the relatively prime condition was placed on parameter CHI. The following treatment uses the optimal value for parameter CHI, 179 TU.

   (b) $RR = \{10, 15, 20, 25\}$

   Passes: 45

Held-constant parameter: $CHI = 179\,\mathrm{TU}$

Held-constant parameter: $CHD = 1$

An analysis of the first treatment for RR shows a gap between $15\,\frac{\mathrm{s}}{\mathrm{rev}}$ to $30\,\frac{\mathrm{s}}{\mathrm{rev}}$ that may produce higher BPS, so the second treatment is intended to explore RR values in that gap.

2. *Focused capture rotation rate* This parameter optimization has a single treatment:

   $FCRR = \{5, 6, 7, 8, 9, 10, 11, 12, 13\}$

   Passes: 30

   Held-constant parameter: $channel = 8$

3. *Channel hop interval* This parameter optimization has a single treatment:

   $CHI = \{109, 119, 129, 139, 149, 159, 169, 179, 189, 199\}$

   Passes: 30

   Held-constant parameter: $RR = 10\,\frac{\mathrm{s}}{\mathrm{rev}}$

   Held-constant parameter: $CHD = 1$

4. *Channel hop distance* This parameter optimization has a single treatment:

   $CHD = \{1, 2, 3, 4, 5\}$

   Passes: 30

   Held-constant parameter: $RR = 20\,\frac{\mathrm{s}}{\mathrm{rev}}$

   Held-constant parameter: $CHI = 179\,\mathrm{TU}$

   This treatment is conducted inside AFIT; specific WAP beacons or a known location is not necessary for this treatment which compares BPS performance across many passes and different CHD.

Table 13. Parameter Discovery Treatments

Parameter	*Treatment*	*Passes*	*Values*	*Held-Constant*
RR	1a	30	$\{5, 10, 15, 30\}$	CHI = 130 TU CHD = 1 ch
	1b	45	$\{10, 15, 20, 25\}$	CHI = 179 TU CHD = 1 ch
FCRR	2	30	$\{5, 6 \ldots 12, 13\}$	channel = 8
CHI	3	30	$\{109, 119, 129 \ldots 199\}$	RR = $10\,\frac{\text{s}}{\text{rev}}$ CHD = 1 ch
CHD	4	30	$\{1, 2, 3, 4, 5\}$	RR = $20\,\frac{\text{s}}{\text{rev}}$ CHI = 179 TU

Positional Captures. After the discovery treatments are performed and optimal parameters are identified, positional capture treatments are conducted. These treatments are identical except for their locations, as specified in Table 12. For these treatments there are no held-constant factors, except for those parameters that have been identified as optimal, which are:

- RR: $20\,\frac{\text{s}}{\text{rev}}$
- CHI: 179 TU
- CHD: 2 ch

5. *Positional Capture 1* Passes: 150
6. *Positional Capture 2* Passes: 150
7. *Positional Capture 3* Passes: 150

The results of these treatments show the performance of localization using these methods and parameters regarding the metrics BE and LE.

Table 14. Positional Capture Treatments

Treatment	*Position*	*Passes*	*Optimal Constant Parameters*		
			RR	*CHI*	*CHD*
5	1				
6	2	30	$20\,\frac{\text{s}}{\text{rev}}$	179 TU	2 ch
7	3				

Focused Captures. This final treatment involves conducting focused captures to identify the parameter FCW that produces the smallest BE.

In designing this treatment, batch captures are programmed to attempt a focused capture on every identified experiment WAP with an FCW of 360 degrees, the largest possible value for FCW. During processing and analysis, each possible FCW is derived from the data set with a real FCW of 360°. The derived FCWs are termed virtual focused capture width (vFCW). For example, a vFCW of 2° is derived that consists of any beacons detected within the 2° bounds. This set is used to produce a bearing prediction, and the accuracy of the bearing prediction is recorded. A new vFCW of 4° is derived, and a bearing prediction is produced for a FCW of 4°. Each possible vFCW is derived (up to a vFCW of 360°) and its bearing prediction performance recorded. The recorded bearing prediction errors are analyzed and an optimal FCW is determined.

For these treatments there are no held-constant factors, except for those parameters that have been identified as optimal, which are:

- RR: $20\,\frac{\text{s}}{\text{rev}}$
- CHI: 179 TU
- CHD: 2 ch
- FCRR: $6\,\frac{\text{s}}{\text{rev}}$

Two treatments are performed at capture locations 1 and 2, per Table 12. These data are used to determine the optimal FCW, as well as demonstrate the LE performance with the discovered optimal parameters.

8. *Focused capture width: Capture 1* Passes: 30

9. *Focused capture width: Capture 2* Passes: 30

4.7.2 Testing Process.

Every capture is conducted at the locations enumerated in Table 12 unless otherwise noted. The process for each treatment is, as follows:

1. The prototype is taken to the appropriate capture location and mounted atop the capture platform securely.

2. Power is provided to both the processor and the motor.

3. Physical connectivity with the processor is established using a laptop with a bridged Ethernet connection and a Cat5 Ethernet cable. The laptop operating system is Windows 10 Enterprise.

4. Network connectivity is established with secure shell (SSH) using Xshell 5. Once established, `tmux` is used to create a terminal session that is maintained if the laptop is disconnected from the processor.

Table 15. Focused Capture Treatments

Treatment	*Position*	*Passes*	*Optimal Constant Parameters*			
			RR	*CHI*	*CHD*	*FCRR*
8 9	1 2	30	$20\,\frac{s}{rev}$	179 TU	2 ch	$6\,\frac{s}{rev}$

5. The antenna is aligned to magnetic north (0° N) using a magnetic map compass.
6. localizer is started in shell mode ($ localizer -s) and batch capture mode is initialized (> batch).
7. The desired capture configuration file is imported (batch> import <conf>).
8. The capture is started (batch> capture).
9. The capture proceeds automatically.
10. Once complete the antenna should be reset North. The bearing is verified to be 0° North, and the capture is validated. If not, the capture is discarded and performed again from step 5.
11. After all captures are completed, the data is committed and pushed to the remote git branch.
12. (Optional) While it is possible to process directly on the capture device, using a more powerful computer provides significantly better processing performance, especially for larger data sets. Pull the captured data from the git repository and process it with the command $ localizer -p. Push the processed files to the remote git branch.
13. Repeat from step 5 as necessary.

4.8 Summary

This chapter outlines the capture process under study in this research, as well as the many necessary details about how the capture method is validated and optimized. The experiment environment is discussed at length, as well as the metrics that are used to judge the effectiveness of the proposed capture process. Parameters that must be optimized are enumerated. Each experiment treatment is discussed in detail. Finally, the explicit testing process is enumerated with necessary detail.

V. Results and Analysis

5.1 Overview

This chapter describes the results of the experiments performed using the `localizer` framework and according to the experimental design in Chapter IV. First, Section 5.2 covers noteworthy observations of the experimentation process. Section 5.3 discusses the findings of the parameter discovery treatments and lists the optimal parameters that are shown in Table 18. The positional capture treatments are analyzed in Section 5.4 with emphasis on the bearing errors measured when using the `localizer` framework to predict the bearing to observed WAPs. Finally, the focused capture treatments are reviewed in Section 5.5 with emphasis on the location errors measured when predicting observed WAP locations.

Post-capture analysis is performed using the Python `pandas` (version `0.22.0`), `scipy` (version `1.0.0`), and `matplotlib` (version `2.1.2`) packages in the JupyterLab `0.30.6` environment.

5.2 Stepper Motor Missteps

The treatments enumerated in Chapter IV were conducted without difficulty except for an issue of missing steps during focused captures that is discussed in this section.

5.2.1 Temperature.

The experimental treatments for focused captures were conducted during a period of cold weather, with temperatures dropping below 0 °C. No prototype equipment was affected by the low temperatures except for the cable lead from the Wi-Fi adapter to the processor. The cable became quite stiff as the PVC sheath became cold.

During a focused capture, the antenna may rotate between $-360°$ and $720°$. As the antenna rotated and the cable wound, the cable's stiffness acted as a spring and resisted movement in both directions. Skipped steps were observed during cold-weather captures due to this effect.

A metal-sheathed cable was obtained to replace the original PVC-sheathed cable, which maintained flexibility well below 0 °C, and the step skipping due to stiff wires was eliminated.

5.2.2 Reset Rate.

As noted in Section 4.4, each capture started and ended with a measurement of the antenna bearing. The measurement served to validate the accuracy of the recorded bearings for each observed beacon; if the final bearing did not match the starting bearing, the stepper motor had missed steps, and the capture set was considered invalidated and discarded. Parameter discovery and positional capture treatments both performed remarkably well, conducting hundreds of captures without missed steps. Unfortunately, the final treatment set (focused capture treatments) suffered from missed steps and invalidated capture sets.

Troubleshooting the issue revealed that cause was isolated to the antenna reset process. Focused captures "reset" the antenna following a capture, preparing for the next capture, moving it from 0 to 180 degrees either clockwise or counter-clockwise based on the optimal path. When resetting, the antenna uses a faster rate of travel, $5\,\frac{\mathrm{s}}{\mathrm{rev}}$. This speed is appropriate when resetting a full revolution (standard for the first two treatment sets), but it causes missed steps when used for very short reset distances. Step skipping was observed at this reset speed between angles of 1 and 30 degrees.

Increasing the $\frac{\text{s}}{\text{rev}}$ (i.e., slowing the reset rate) to a value that would be safe for very low rotational distances, would increase the capture time to an unacceptable level. A solution was determined in the form of a variable reset rate based on the reversed sigmoid function

$$S(x) = a + \frac{b - a}{1 + \left(\frac{x}{c}\right)^d} \tag{10}$$

where values for a, b, c, and d were found using non-linear least squares interpolation (from the Python `scipy.optimize.curve_fit` library) and the following initial values:

$$y = \{20, 7, 5, 4\}$$
$$x = \{0, 90, 180, 360\}$$

The initial values indicate that at a rotation distance of 0°, the reset rate should be 20 $\frac{\text{s}}{\text{rev}}$, at 90° it should be 7 $\frac{\text{s}}{\text{rev}}$, and so on. The least squares optimization produces the coefficients

$$a = 3.235$$
$$b = 20.000$$
$$c = 34.681$$
$$d = 1.300$$

Reset rotation rate RR_r as a function of rotation distance δ, is shown in Figure 16 and given as:

$$RR_r(\delta) = 3.235 + \frac{16.765}{1 + \left(\frac{\delta}{34.681}\right)^{1.300}} \tag{11}$$

The sigmoid model script is found in Appendix C.1.

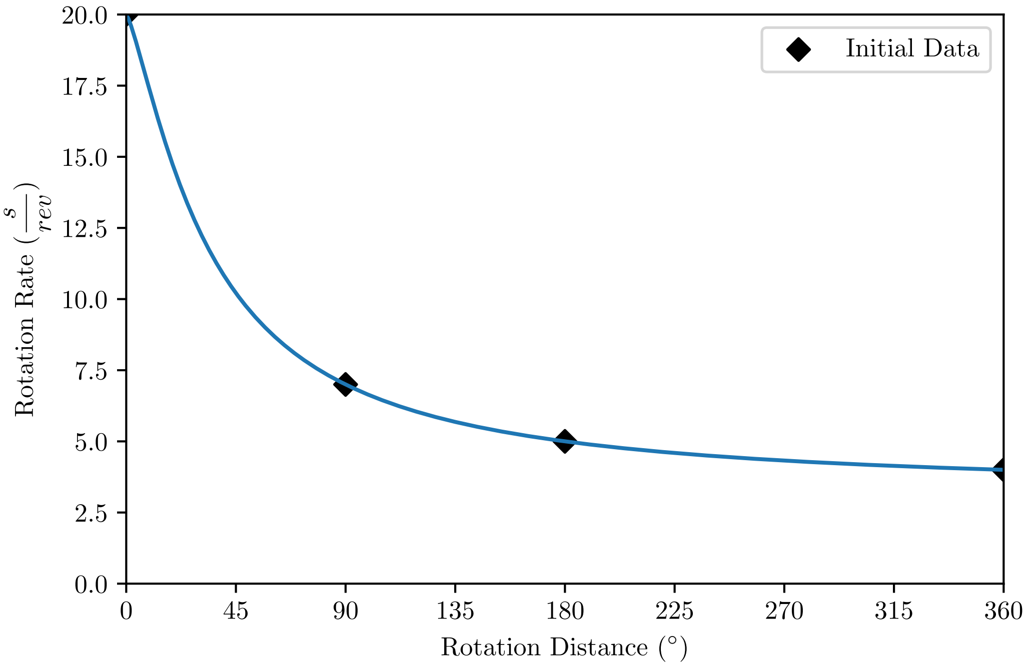


Figure 16. Reset Rotation Rate RR_r (Sigmoid)

5.3 Parameter Discovery Analysis

Parameter discovery treatments provide data that is analyzed in this section to indicate ideal capture parameters. This section summarizes the findings of treatments 1-4, per Table 13, where all held-constant factors are listed.

5.3.1 Rotation rate.

Rotation rate consists of two treatments, a preliminary treatment that identified the need for further data, and the follow-on treatment to collect it. The intent of these is to discover the RR value that optimizes BPS.

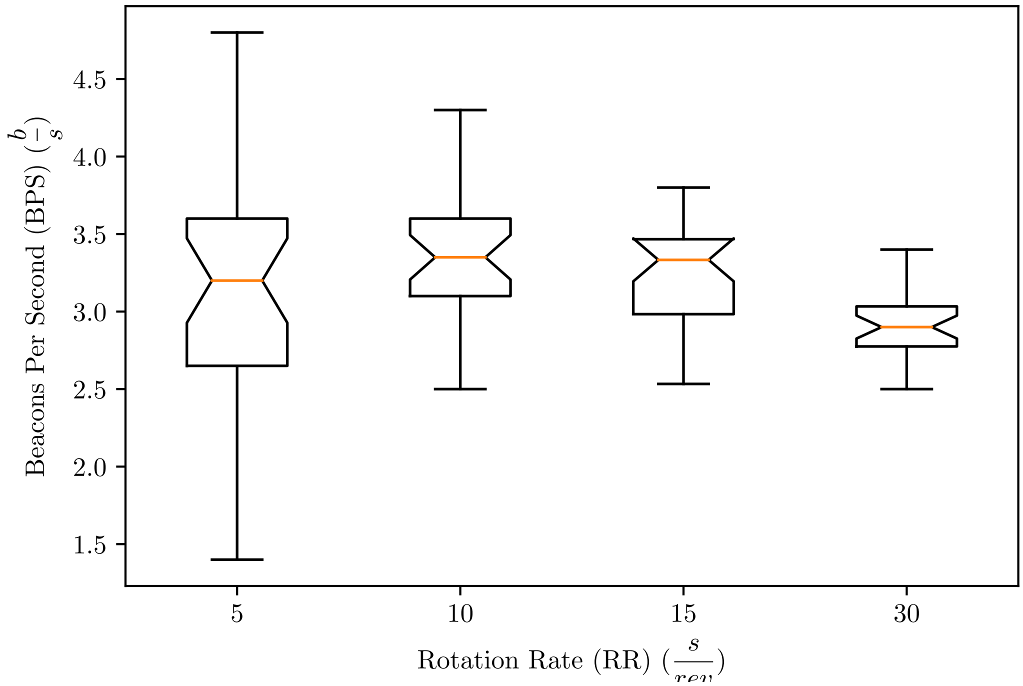
Treatment 1a. This treatment iterates over four values $\{5, 10, 15, 30\}$ of 30 passes each to produce the box plots of Figure 17a. The whiskers of the plot show the tendency of the BPS to normalize over longer captures as evidenced by the lower variance of longer captures. These results indicate that there may be more optimal values between 10 and 30 $\frac{s}{rev}$, which is the impetus for RR's additional treatment.

Treatment 1b. This treatment bridged the gap between 10 $\frac{s}{rev}$ to 30 $\frac{s}{rev}$, testing the values $\{10, 15, 20, 25\}$ over 45 passes. The results are displayed in Figure 17b with the optimal value for CD highlighted, 20 $\frac{s}{rev}$. The highest median BPS of 3.45 $\frac{b}{s}$ is slightly higher than our maximum expected value of 3.26 $\frac{b}{s}$, likely due to an abundance of reflections and cross-channel observations. The notches of the boxes indicate the 95% confidence interval of the data, with the optimal RR value being particularly tight relative to the other parameter values [28].

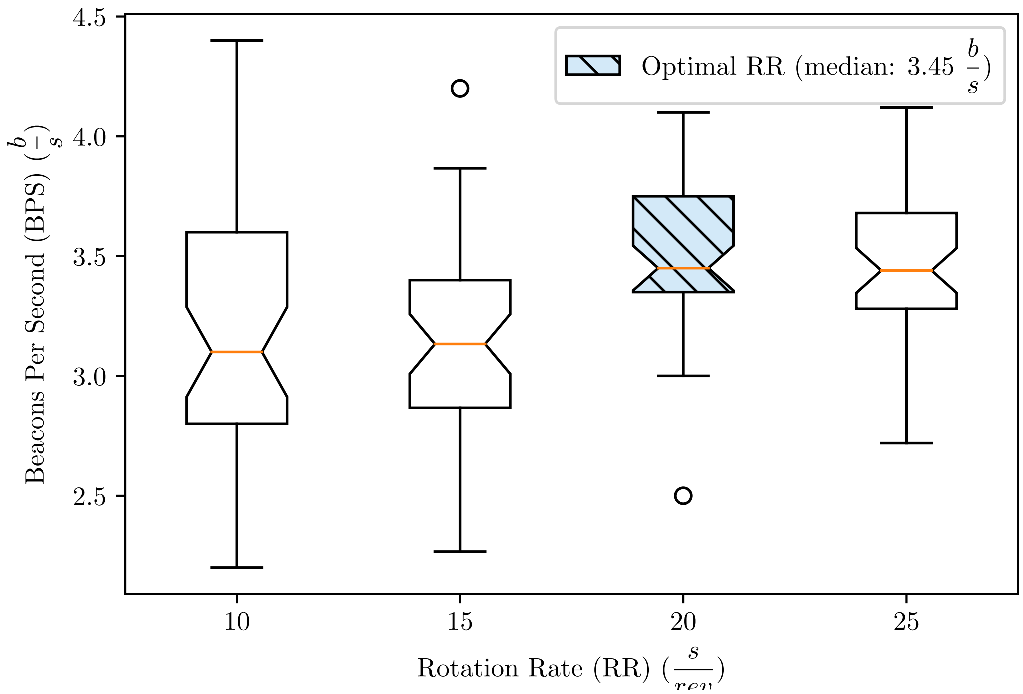
5.3.2 Focused capture rotation rate.

Treatment 2. This parameter received only a single treatment, holding the channel constant at FCRRs of $\{5, 6, 7, 8, 9, 0, 11, 12, 13\}$, each over 30 passes. The optimal parameter is 6 $\frac{s}{rev}$, which produces a median BPS_f of 10.17 $\frac{b}{s}$ shown in Figure 18.

The median value of FCRR at 6 $\frac{s}{rev}$ is surprising, given that it exceeds the expected maximum value of BPS of 9.77 $\frac{b}{s}$ estimated in Section 4.5. The likely reason that this value is so high is due to the high amount of reflections in the capture environment. Another possibility is the extra amount of time that CaptureThread captures packets before the timer starts on the given CD.



(a) Treatment 1a Results



(b) Treatment 1b Results

Figure 17. Rotation Rate (RR) Treatment Results

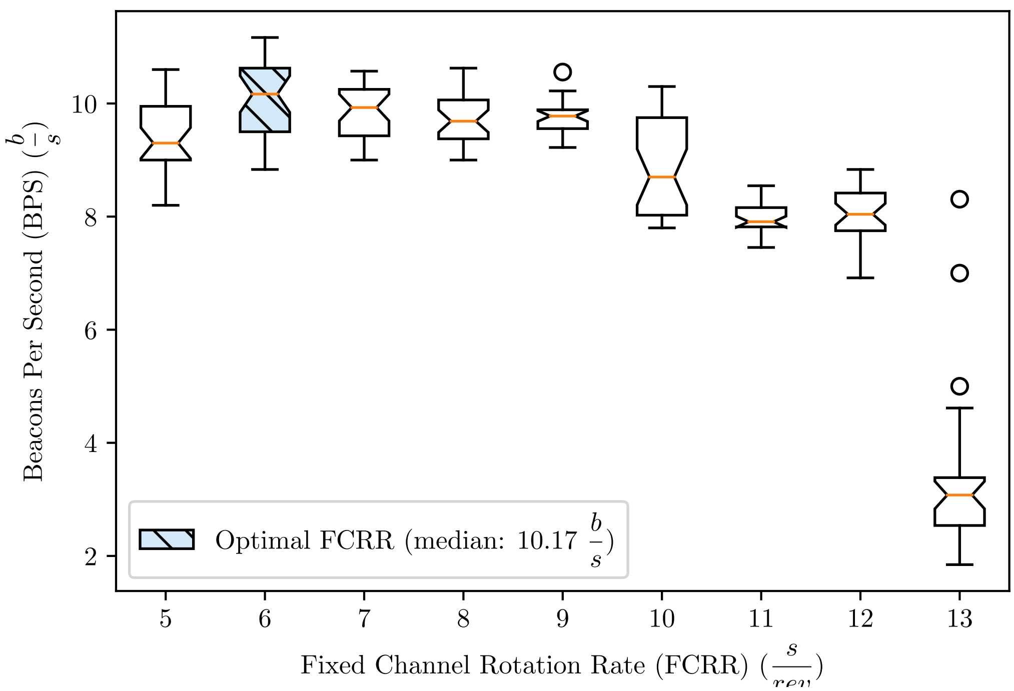


Figure 18. Focused Capture Rotation Rate (FCRR) Treatment Results

5.3.3 Channel hop interval.

Treatment 3. This parameter was tested at the values of {109, 119, 129, 139, 149, 159, 169, 179, 189, 199} over 30 passes. The optimal BPS is 3.75 $\frac{\text{s}}{\text{rev}}$, discovered at parameter value of 179 TU as shown in Figure 19.

5.3.4 Channel hop distance.

Treatment 4. This parameter was tested at the values of $\{1, 2, 3, 4, 5\}$ over 30 passes. The optimal BPS is 3.40 $\frac{\text{s}}{\text{rev}}$, discovered at parameter value of 2 ch as shown in Figure 20. 2 ch has more variance than 1 ch, which is speculated to be caused by the occurrence of observations across channels, such as observing channel 5's beacons on channels 4 and 6.

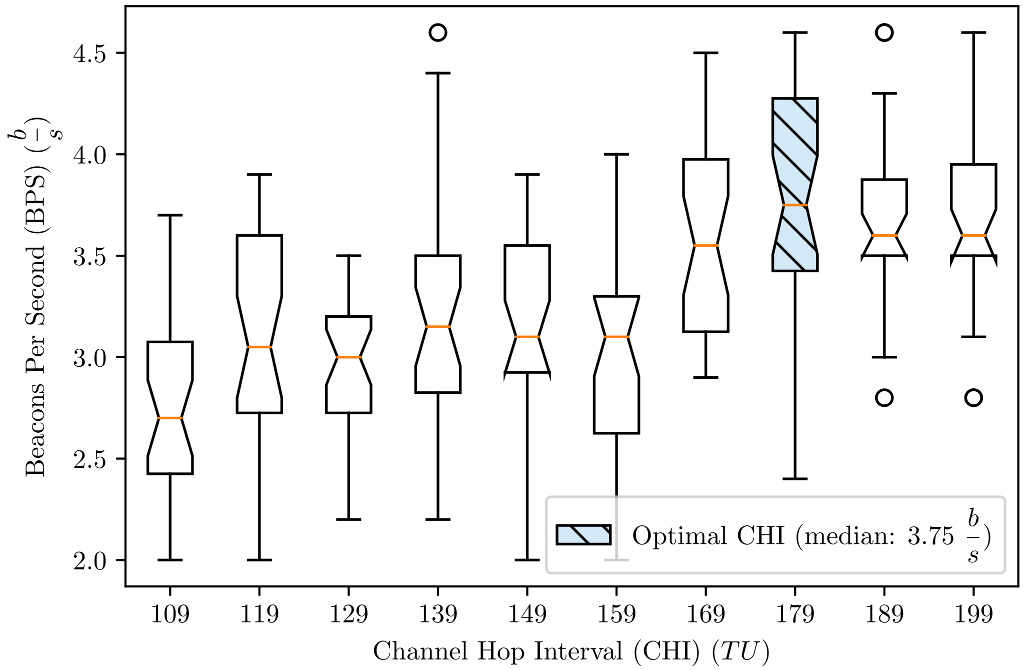


Figure 19. Channel Hop Interval (CHI) Treatment Results

5.4 Positional Capture Analysis

Treatments 5-7. Positional capture treatments provide data that is analyzed in this section to test the performance of the optimal parameters identified in the previous section. Each treatment is processed to determine the BE as defined in Section 4.5, by doing the following:

1. **Extract Series.** For each capture across all three treatments, create a series of the beacon RSSI values as a function of bearing for each unique BSSID.

2. **Convert RSSI (dBm) to mW scale.** Converting dBm to mW produces improved prediction accuracy by penalizing very low RSSI over relatively higher RSSI values.

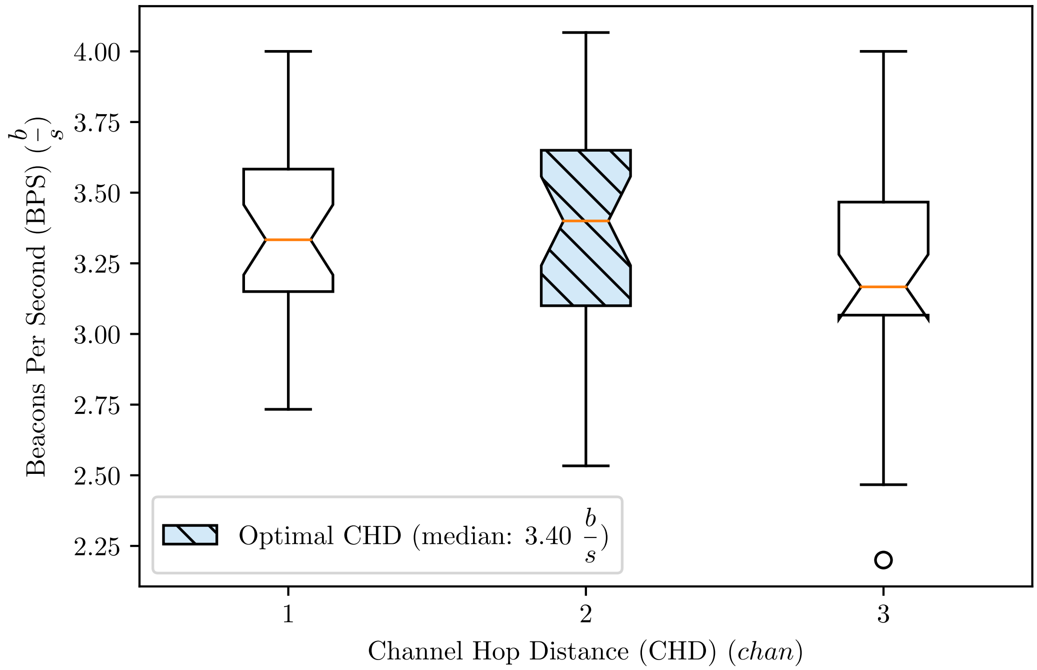


Figure 20. Channel Hop Distance (CHD) Treatment Results

3. **Interpolate.** For each series, interpolate the sparse data using all valid interpolation methods provided by the `scipy.interpolate` library.

4. **Calculate Error.** Determine the error of each interpolated series and measure it against the true bearing to determine BE.

5. **Record Values.** Record the values to a table for analysis.

5.4.1 Interpolation.

To determine the best interpolation method, the median BE is determined for all interpolation methods provided by `pandas.series.interpolate` [29] (`scipy.interpolate` [30]). To compute these values, each interpolation method is performed on each series in the data captured in treatments 5-7, with 4,342 total series derived from the treatments. The prediction is performed for each and compared with truth, and an error value is determined and stored. The median of these errors is displayed in Table 16 sorted by performance in ascending order.

Table 16. Interpolation Performance

Method	*Median Error*
PCHIP	13.70°
BPoly	14.31°
Naive	14.70°
SLinear	14.83°
Linear	15.09°
Akima	16.44°
Bayercentric	22.83°
Cubic	24.94°
Quadratic	25.70°
Krogh	59.28°
Random	89.68°

The following are the top performing interpolation methods with their median error rate for all sample sizes and a brief description of the interpolation technique employed:

1. **Piecewise Cubic Hermite Interpolating Polynomial (PCHIP)**

   Error: 13.70°

   This interpolation method is a spline where each interval is limited to a third-degree polynomial and produces a smooth, continuous function with a continuous first derivative, a feature of the Hermite form.

2. **Bernstein Polynomial (BPoly)**

   Error: 14.31°

   This interpolation method is a spline where each interval is Bernstein basis polynomial, a limited polynomial form that eases approximations such as continuous interpolation.

3. **Naive**

   Error: 14.70°

   The 'Naive' method is not an interpolation method, but the simplest method of choosing a bearing by simply selecting the bearing with the highest mW value.

The interpolation and bearing prediction process is illustrated in Figure 21 where Piecewise Cubic Hermite Interpolating Polynomial and Bernstein Polynomial methods are performed on the same sparse beacon series. The vertical lines represent the interpolation methods' predictions as well as the true bearing to the WAP.

To visualize the performance of each interpolation method, Figure 22 shows the top three interpolation methods as functions of beacon series sample size.

5.4.2 Bearing Error Analysis.

Further analysis shows that the performance of the interpolation methods varies based on how many beacons that are observed per capture, demonstrated by Figure 22. Except for the case of a sample size of 1, where SLinear performed the best with a median BE of 26.75°, nearly all the rest of the sample sizes performed best with the PCHIP interpolation method, which had an overall median BE of 13.70°, significantly lower than our expected maximum of 45°, per Table 9.

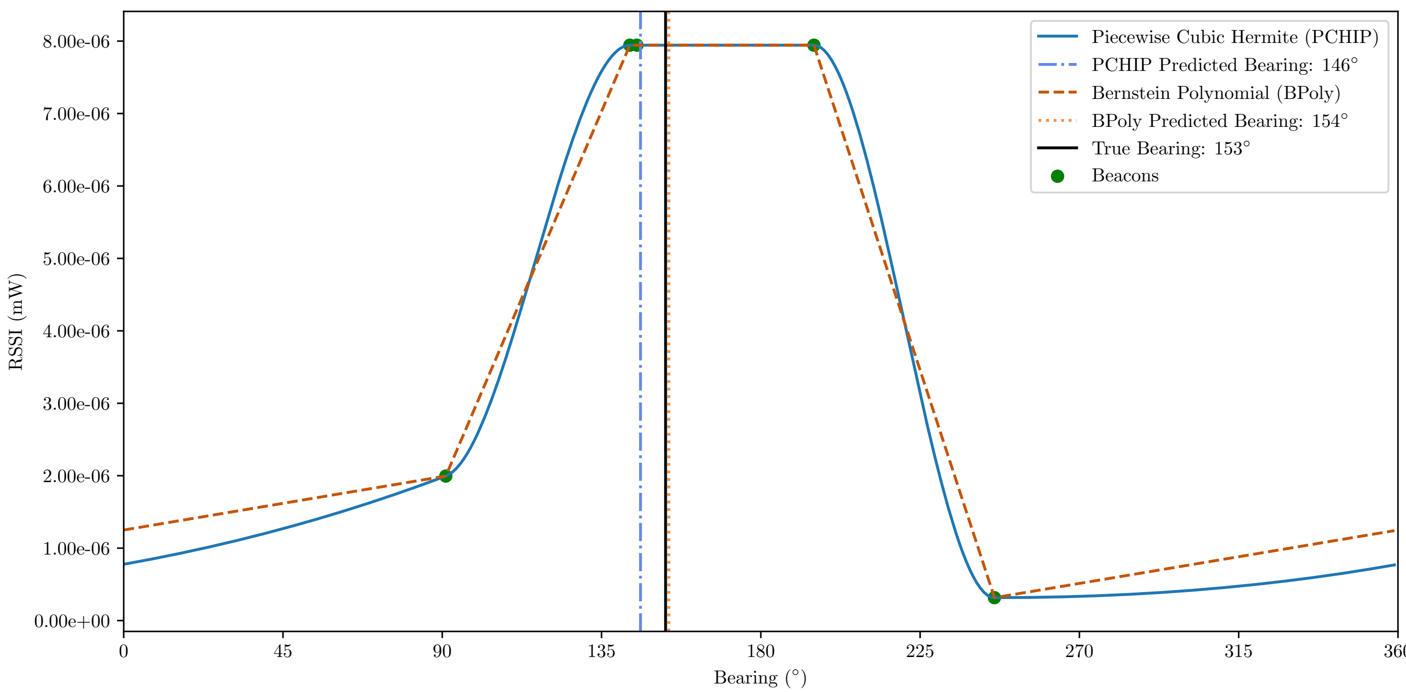


Figure 21. Interpolation of 5-Sample Capture using PCHIP and BPoly Interpolation Methods

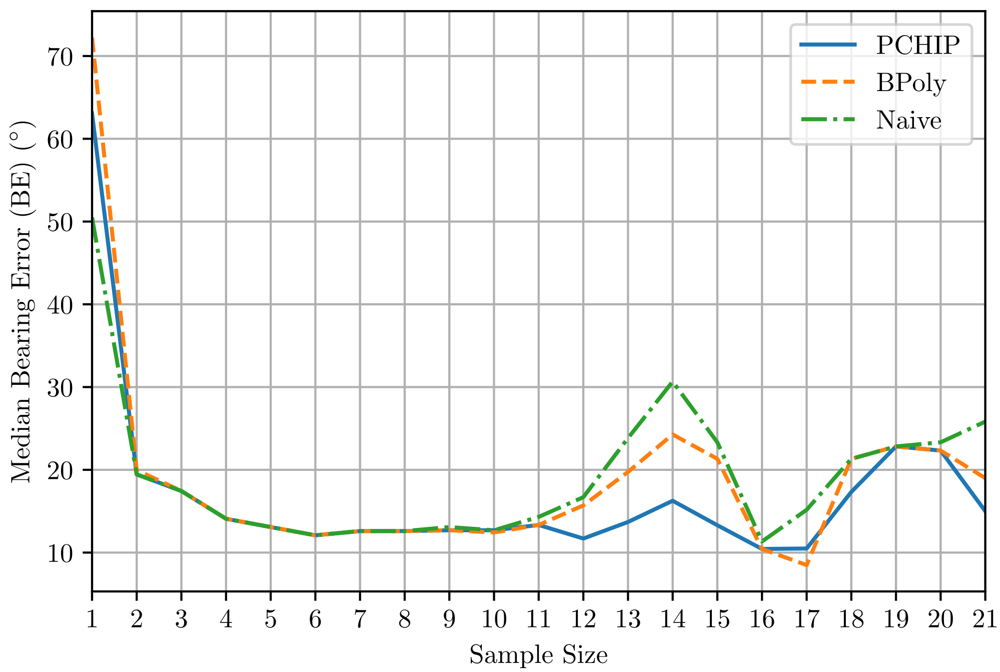


Figure 22. Interpolation Performance Per Beacon Sample Size

Figure 22 shows a steady reduction and smoothing in bearing error as the sample size increases, however the error increases at 14 and 15 beacon sample sizes for an unknown reason. Analysis of the data did not produce clues as to the cause of this bump, but the disparity of th pchip interpolation and naive method are demonstrated here, where an interpolation method does demonstrate significant performance advantage over the computationally simpler naive method. The increase in bearing prediction error in the larger beacon sample sizes, such as 19 and 20 are due to the significantly smaller number of series for those particular sample sizes, as demonstrated in Figure 25.

The performance of PCHIP overall is summed up in Figure 23, which shows the PCHIP BE box and histogram charts. While the median is quite close to zero, there are significant outliers throughout, represented in the box plot by the black ticks outside the box. There are peaks at 180° and −180° visible at both ends of the histogram which represent a high number of errors at 180°.

It should be noted that bearing errors are calculated using the absolute value of the difference between true and predicted bearings. Otherwise, the median errors would be misleadingly close to 0. It is useful, however, to plot the first, second, and third quartiles of PCHIP BE to observe the spread of BEs, shown in Figure 24. This Figure shows the high accuracy of the wide sweep when at least 2 or 3 beacons are observed for a given BSSID. Figure 25 shows the number of sets per sample size to show the significance of the data in Figure 24.

The top interpolation method for each recorded sample size is given in Table 19 in Appendix E.

Figures 32, 33, and 34 in Appendix E show the performance of PCHIP for each BSSID as polar histograms. The figures for treatments 5 and 7 show a notable 180° error for WAP RESEARCH_MULLINS_7; treatment 7 for this WAP is shown in Figure 26. Looking at the map (Figure 14) it can be surmised that these errors are due to reflections off the building directly behind the prototype at capture points 1 and 3. These errors also contribute to the outliers at 180° and −180° in Figure 23.

5.4.3 Location Error Analysis.

Bearing predictions from multiple captures may be combined to predict the location of a WAP. When the predictions are represented as a ray with an origin component and vector component, the point nearest to all rays is presumably the point closest to the emitter.

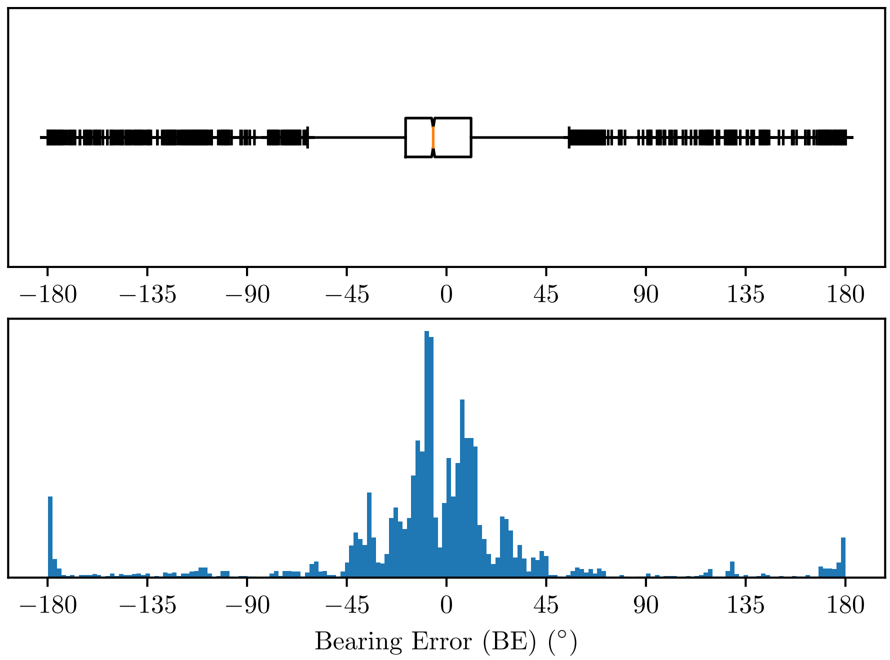


Figure 23. PCHIP Error Statistics

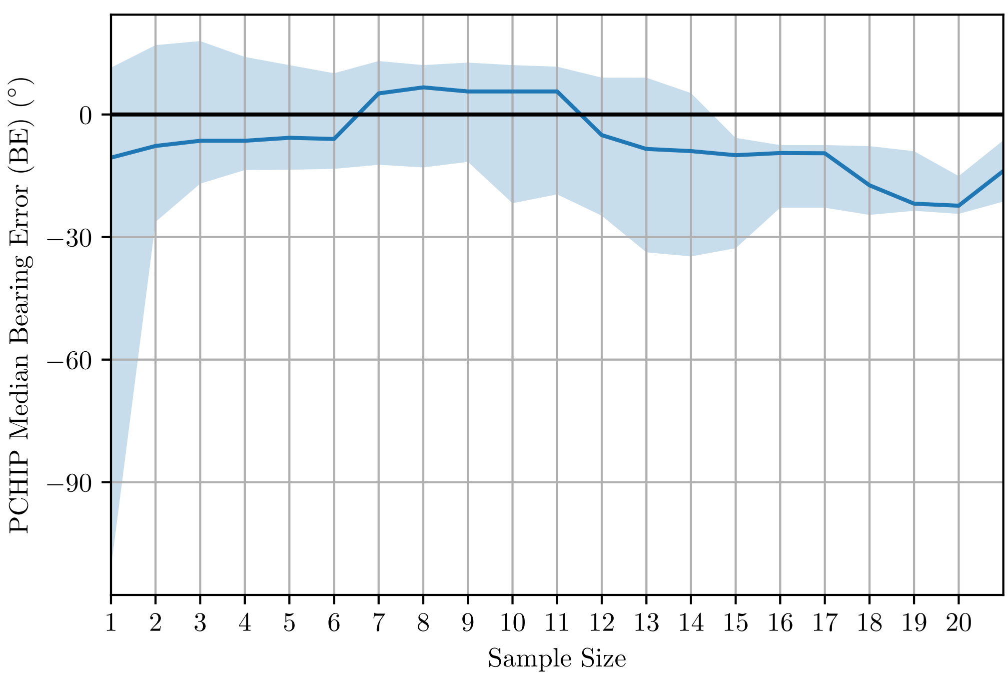


Figure 24. PCHIP Interpolation Quartiles By Beacon Sample Size

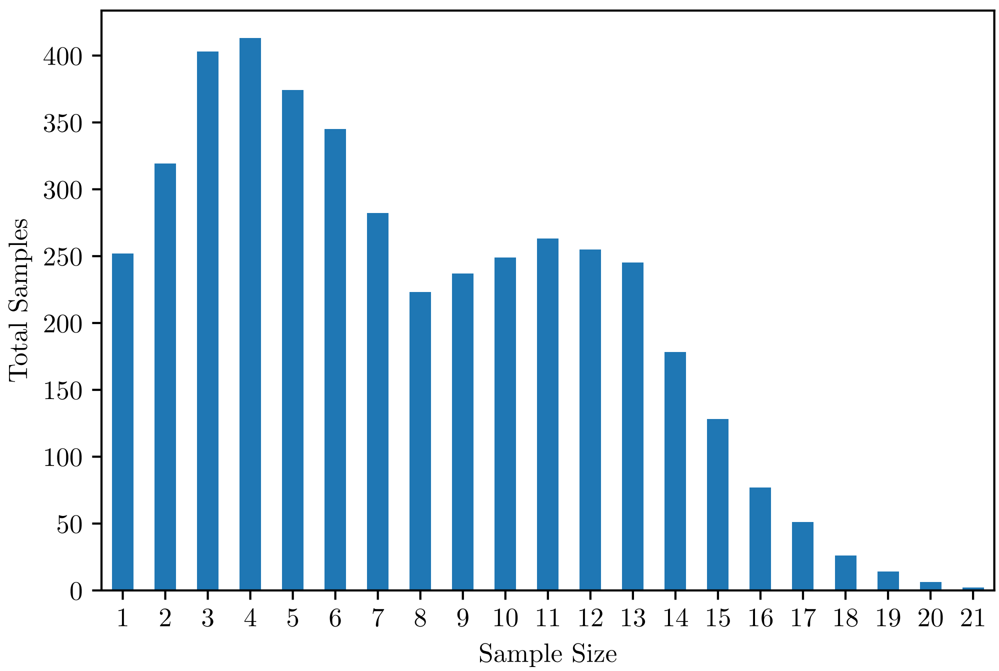


Figure 25. Interpolation Series Per Beacon Sample Size

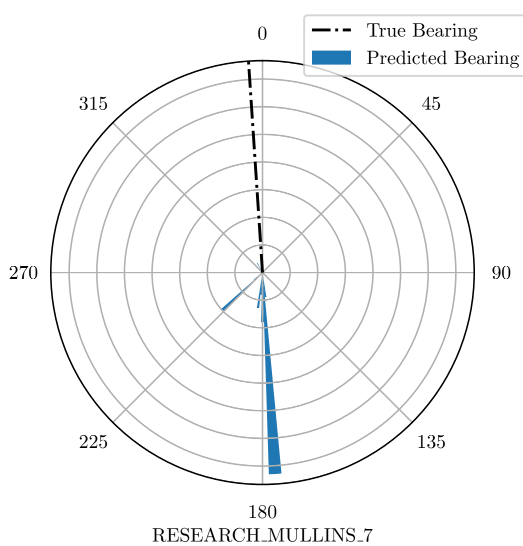


Figure 26. PCHIP Polar Prediction Histogram

This research uses the least squares optimization implemented in Appendix F where any number of rays may be processed and a point nearest all of them is produced. Least squares optimization is performed using data from treatments 5, 6, and 7 for each combination of capture location, capture pass, and BSSID, totaling 5,848,250 combinations. Because many captures did not receive beacons from all 10 WAPs, the total number of location prediction sets is reduced to 5,319,396.

Combinations of two capture locations are compared to combinations with three in Figure 27 to determine if there is a significant advantage to adding a third bearing prediction.

There are significant large outliers in these results, caused by very nearly parallel bearing predictions that converge hundreds or thousands of meters away. Furthermore, when all rays diverge, the closest point to the rays is the mean of their origins, which in this research is never the correct location.

The results from the least squared error localization method are higher than expected, with median location errors over 10 m higher than our expected maximum of 50 m. The three capture sets performed notably better than the two capture sets, with median errors of 69.93 m and 60.66 m respectively.

Two examples of the results of the least squares optimization function are displayed in Figures 28a and 28b, demonstrating low location error and high location error, respectively. Figure 28c shows an example of a case of reflection that contributes to the very highest of location prediction errors - each bearing prediction is in the opposite direction due to reflections off the surrounding structures. This is an example of a limitation of the experiment environment that will not exist on an airborne prototype.

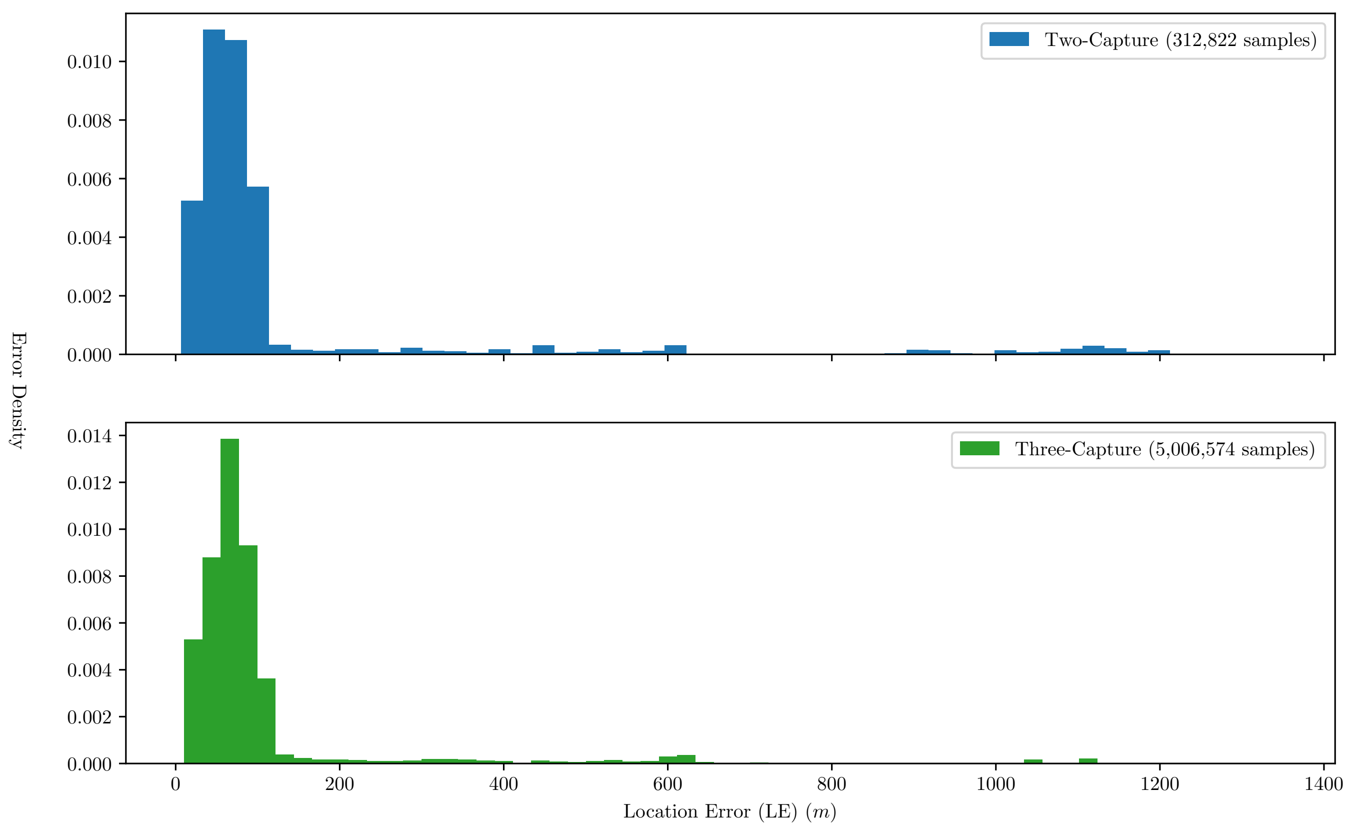


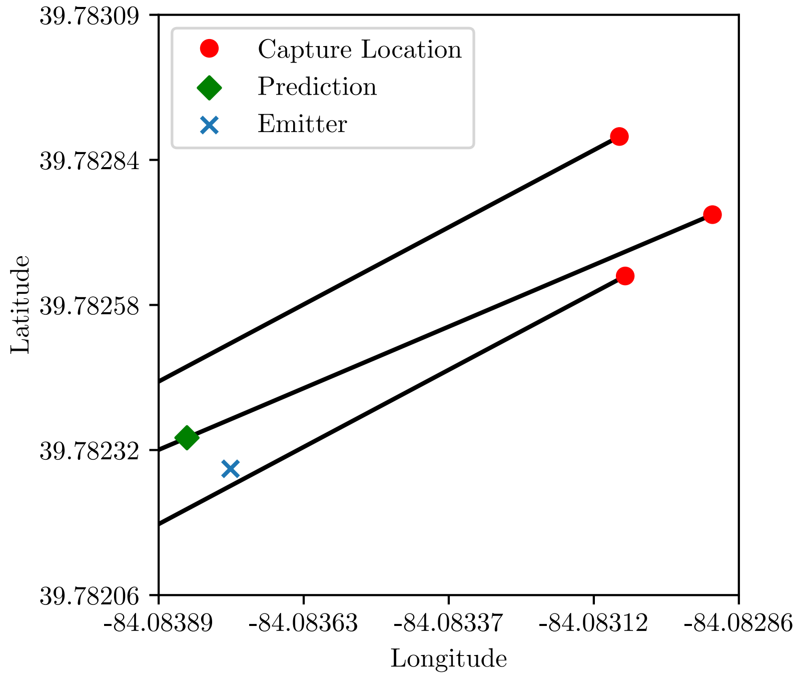
Figure 27. Location Errors for Capture Sets of Two and Three Locations

High Location Error Analysis and Compensation. The analysis of bearing prediction performance in Section 5.4.2 shows that bearing predictions can be very accurate, however even small errors in bearing prediction may severely hinder location prediction performance if the bearing predictions diverge. When the bearing predictions diverge, then the most "optimal" location prediction is the mean position of each ray starting point, a significantly high location error in these experiments. Additionally, if the bearing predictions converge but are very close to parallel, then the location prediction may be very distant at their point of intersection (see Figure 28b). Limits may serve to counter these cases and improve location prediction performance.

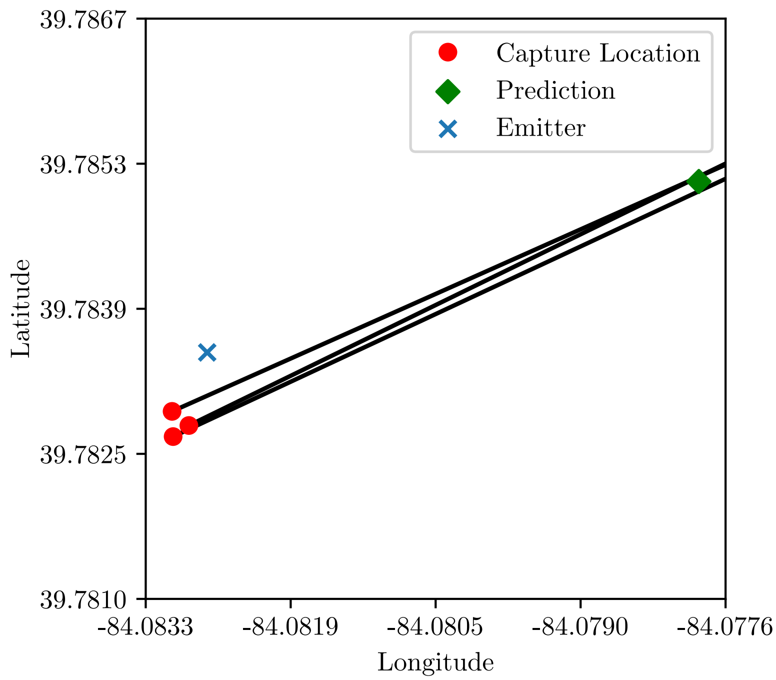
To test this theory, the data was filtered to eliminate those predictions with an error of greater than 500 m and those with a prediction that is less than 1 m from the mean position of the prediction rays, as enumerated in Table 17. The number of predictions of the original 5.3 million that remain is 4.4 million, which indicates a reduction of 17%, the majority of those eliminated due to the 1 m mean distance from ray origin constraint (13%). Figure 27 is reproduced as Figure 29 to show the difference made by constraints. With these constraints applied, median error for two-ray and three-ray predictions remain practically unchanged at 69.14 m and 60.14 m respectively.

Table 17. Location Constraints

Metric	*Constraint*
Distance from mean ray origin	>1 m
Distance from true emitter location	<500 m

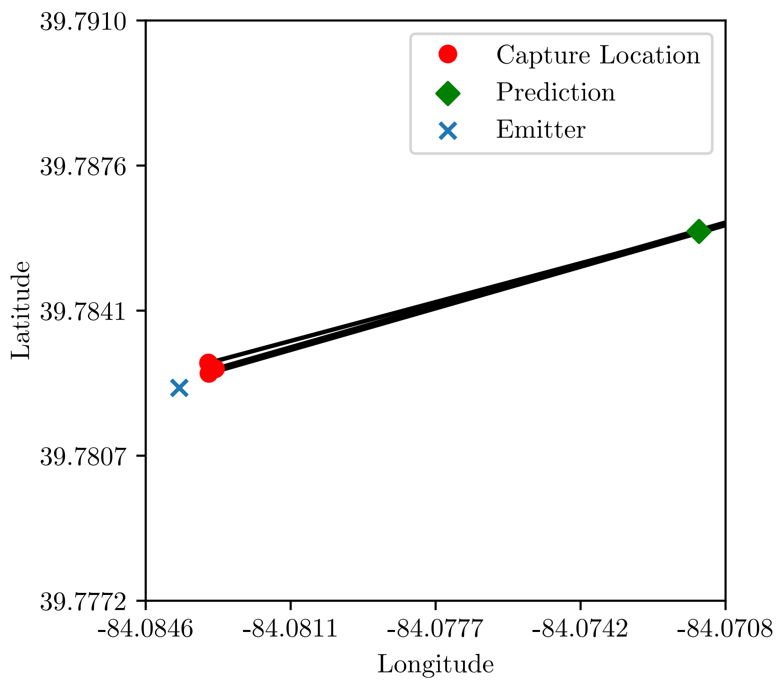


(a) Low Location Error Example (Error: 9.24 m**)**



(b) High Location Error Example (Error: 453.78 m**)**

Figure 28. Location Error Examples



(c) Reflection Error Example (Error: 1.13 km**)**

Figure 28. Location Error Examples (cont.)

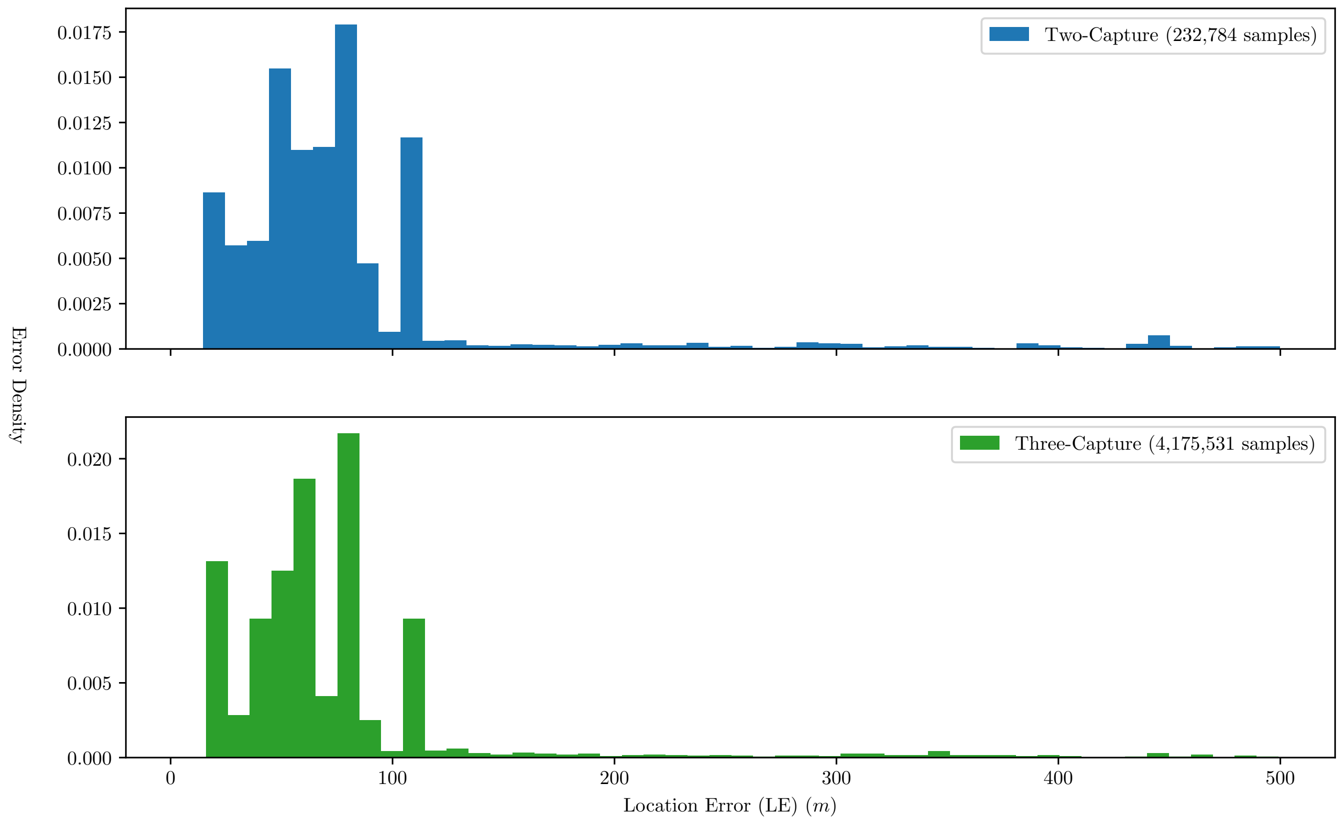


Figure 29. Location Errors for 2 and 3 Captures With Constraints

It is possible that improved location prediction may be achieved by using a weighted algorithm to penalize very distant predictions or to assign lengths to the ray vector component that is relative to the PR value. This would serve to avoid very distant prediction errors, however, Figure 29 shows that those outliers are not common and do not affect the median error to a significant degree.

5.5 Focused Capture Analysis

5.5.1 Focused capture width.

Treatments 5-7. The data from these treatments identify the optimal FCW. Interestingly, the best performing FCW barely outperformed the median PCHIP BE rate, only improving the wide capture performance by 1°.

Figure 30 shows the first and third quartiles, in addition to the gap between them as a function of FCW.

The gap levels out near 90°, where the median is also very close to zero. The optimal FCW near 90° is identified as 84°. Detailed statistics for this FCW are shown in Figure 31, and nearly identical standard deviation of 56.44° when compared with wide capture performance (Figure 23).

5.5.2 Focused Capture Analysis Summary.

Surprisingly, focused captures do not provide significantly lower BE. It is likely not necessary to perform a focused capture, except in cases where a wide capture failed to produce a good bearing (i.e., the DWAP could not connect to the WAP at the predicted bearing). If it is necessary, an FCW of 84° is optimal.

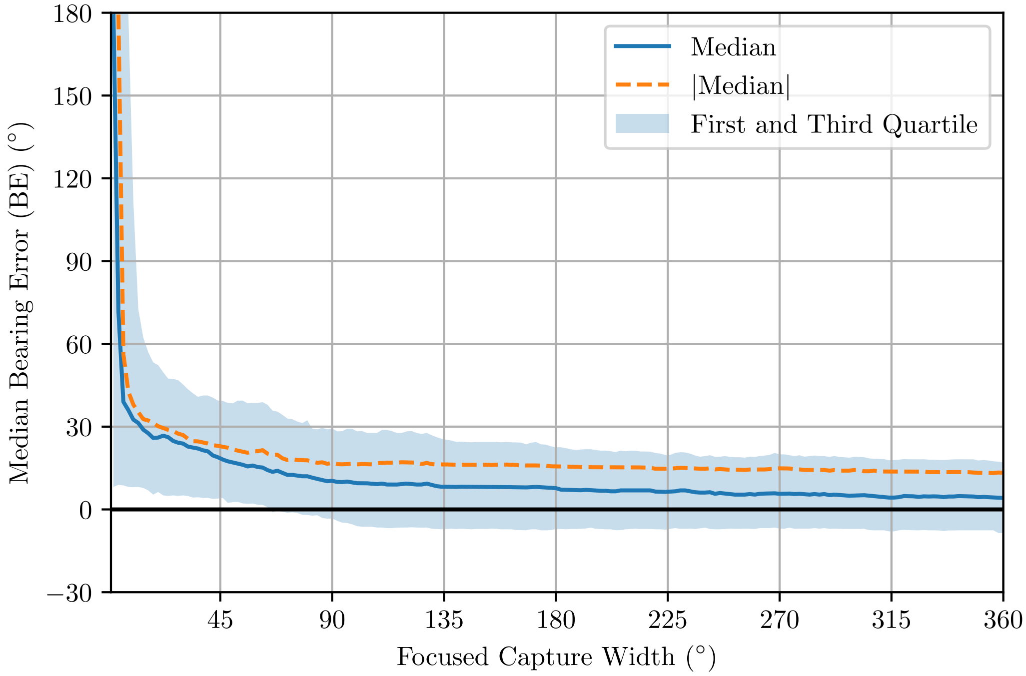


Figure 30. Bearing Error as a Function of Focused Capture Width

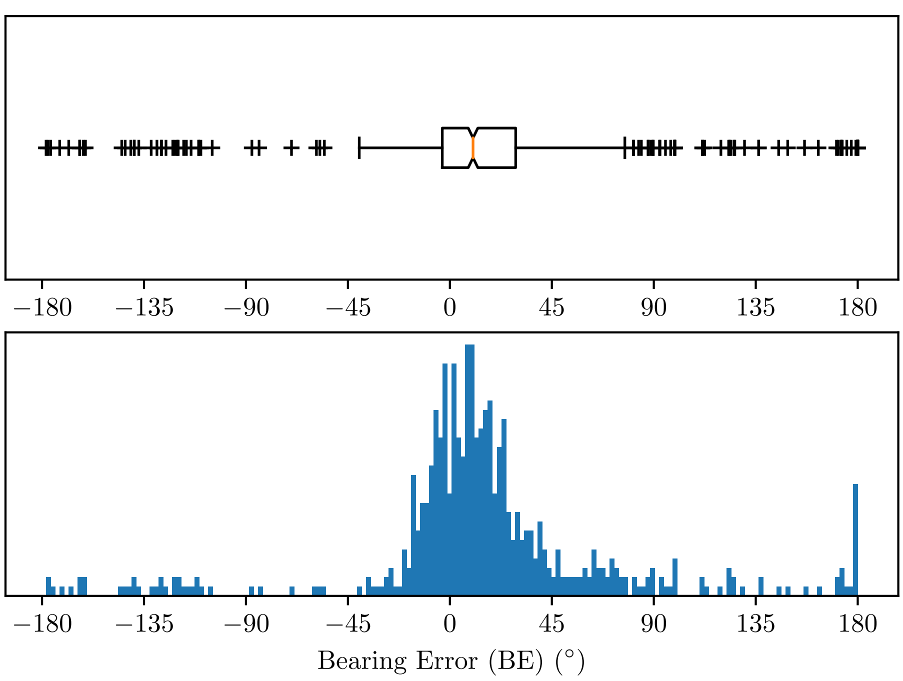


Figure 31. Bearing Error Histogram with a Focused Capture Width of 84°

5.6 Analysis Summary

Table 18 summarizes the discovered parameters that produced optimal bearing and location predictions.

The performance of wide captures using these parameters concerning BE is on the low end of the expected value range in Table 9. Unexpectedly, focused captures failed to produce significantly better BE performance indicating that wide captures are sufficient for most purposes. Focused captures may still be useful for certain applications.

Location prediction performance is lower than expected, yet future work may be able to optimize the parameters further, such as capture distance, and by introducing more rays to the optimization function.

Table 18. Optimal Parameters

Parameter	*Units*	*Optimal Value*
Rotation rate (RR)	$\frac{s}{rev}$	$20\,\frac{s}{rev}$
Focused capture rotation rate (FCRR)	$\frac{s}{rev}$	$6\,\frac{s}{rev}$
Channel hop interval (CHI)	tu	179 TU
Channel hop distance (CHD)	ch	2 ch
Focused capture width (FCW)	°	84°

VI. Discussion and Conclusion

6.1 Overview

UAV technology is evolving at a rapid pace, and the consumer market is producing UAVs that fly farther, operate longer, carry more weight, and cost less than the previous generation. This trend shows few signs of slowing, and network administrators and defenders must understand the threats they face in this area.

UAVs present threats in the form of DWAPs, particularly because of the availability and low cost of the parts necessary to construct one. This research successfully demonstrated how a hypothetical DWAP, equipped with a high-gain directional antenna, might identify and locate target DWAPs, and also showed that inexpensive, low-performance hardware is capable of predicting target WAP bearings.

This chapter summarizes the research and analysis from this experimental work. Section 6.2 summarizes conclusions drawn from the experimental results. Section 6.3 highlights the significance of the research findings. Finally, Section 6.4 enumerates possible expansions of this research.

6.2 Research Conclusions

The goals of this research were to develop passive WAP bearing and location prediction techniques designed to work on an airborne platform. To fulfill these goals, a hardware prototype was designed and built that fulfilled the goals of low weight, low cost, and adequate processing power for the required tasks. A software framework, `localizer`, was purpose-built for the prototype to perform data collection and real-time bearing prediction.

The hypothesis that a DWAP-mounted directional antenna may be used to identify the bearing and location of a WAP was tested by first identifying optimal parameters for data capture. Optimal values for all workload parameters listed in the System Under Test (Figure 13) were discovered and data capture was conducted to collect a large body of data. The captured data was analyzed to measure performance of different bearing and location prediction methods.

Analysis of the data captured by the `localizer` framework shows that the PCHIP interpolation method produces median bearing predictions as low as 8.49°, and a median prediction error of 13.70° for all sample sizes. The bearing prediction hypothesis was proven in this research, and a simple method of reliably performing bearing predictions is demonstrated. The focused capture method did not produce meaningful improvements in bearing accuracy, which was unexpected, however, this result affirms the strength of the wide capture method. Furthermore, the workload parameters produced by this research provide a starting point for future research into the viability of using directional antennae on UAV platforms.

The location prediction method of using least-squares optimization of multiple bearing predictions performed worse than expected and does not represent a good localization technique in its current state. This research failed to prove this hypothesis, but the results nonetheless present a valuable insight into using least-squares optimization to derive an emitter's location based on multiple bearing recordings.

The goals of this research were met with the development, testing, and validation of a hardware prototype, software framework, and successful bearing prediction technique. Furthermore, the failed location prediction hypothesis provides insight and a starting point for future research in that area.

6.3 Research Significance

If DWAPs are to employ directional antennae and benefit from the benefits that directionality provides, they must necessarily overcome the problem of target bearing radiolocation. This research provides a feasible methodology to overcome this limitation, as well as a software framework for implementing it and conducting future research.

The bearing prediction performance that is demonstrated in this research is sufficient to facilitate target detection, connection, approach, and attack. These capabilities are a stepping-stone to implementing long-range directional DWAPs that are capable of tracking targets and performing wireless CNO over large areas.

Even though the location prediction performed poorly, there is value in demonstrating the performance and limitations of performing location prediction using triangulation and least-squares optimization of bearing predictions.

6.4 Future Work

This research builds a foundation for significant future research in the DWAP field. Possible future research may include:

- **Airborne Performance.** The most direct extension of this research is to move the prototype to a UAV platform and conduct similar experiments to determine how much, if any, the bearing and location predictions improve.

- **Intelligent RSSI Filtering.** Eliminating beacons with RSSI values that are small, relative to the rest of the captured beacons, could filter out reflected RSSI readings. Another approach could include making multiple predictions if there are high RSSI values at bearings that are significantly different.

- **Expanded Capture Positions for Location Prediction.** Location prediction using the least-squares approach described in Section 5.4.3 is expected to have higher accuracy if the data is captured from positions that surround the targets; this approach is likely to cut the large error predictions down substantially.

- **Antenna Testing.** Different directional antennae have different radiation patters and perform differently. Identifying the optimal antenna configuration for a DWAP may be done by profiling the performance of multiple antenna candidates.

- **Mobile Target Tracking.** With improvements in location prediction, mobile Wi-Fi emitters may be tracked by continually generating bearing predictions from different locations. Probe requests may also be included to track and profile smartphones from a long distance and over a wide area.

- **Swarming.** Multiple DWAPs may conduct captures and generate independent predictions which, when shared between them, may produce near real-time location predictions. Other methods of localization, such as AOA, TOA, and TDOA may be used to improve location predictions.

- **Autonomy.** An autonomous DWAP may conduct localizing functions within predetermined parameters, such as remaining within a certain radius, seeking out unsecured or low-secured WAPs, conducting automated site surveys, or following a particular smartphone. With the many applications of UAV and DWAP autonomy, research is necessary to determine how these functions might best be performed.

Appendix A. `localizer` Manual

Common between nearly all modules is module-tagged logging initialization, allowing for simple and detailed logging throughout code execution. Logging occurs to a `localizer.log` file, as well as to the console when executing in interactive mode.

A.1 Initial Installation

Before `localizer` can be used, it must be installed using the provided `setup.py` installation script. If Python 3.5 and `pip` are installed, installing `localizer` is performed typing the following code in the directory with `setup.py`:

```
$ pip install .
```

`localizer` requires root to manage the different capture modules, such as GPS, GPIO, and setting the Wi-Fi interface in monitor mode. Once `localizer` and its dependencies are installed, the help command shows what available program arguments are available.

```
# localizer -h
usage: localizer [-h] [-d] [-w WORKINGDIR] [-p] [-m MACS] [-ccw] [-s]
[--serve]

optional arguments:
-h, --help            show this help message and exit
-d, --debug           Make debug output print to the console. This flag may
also be set in the shell
-w WORKINGDIR, --workingdir WORKINGDIR
Set the parent directory for session experiments. If
blank, current directory is used.
-p, --process         Process the files in the current directory, or a
provided working directory (-w)
```

```
-m MACS, --macs MACS  If processing, a file containing mac addresses to
filter on
-ccw, --counterclockwise
Set this flag if the captures were performed in a
counter-clockwise direction
-s, --shell           Start the localizer shell
--serve               Serve files from the working directory on port 80.
This flag may also be set in the shell
```

A.2 Interactive Shell

An operator enters the `localizer` shell by the command `localizer -s`, after which they are greeted by an interactive prompt

```
$ localizer -s
Welcome to Localizer Shell...
/root:2018011..:wlan0:15s>
```

The shell has many commands that are listed with the `help` command

```
/root:2018011..:wlan0:15s> help

Documented commands (type help <topic>):
========================================
batch    cd     exit  help  process  serve  shell
capture  debug  get   list  quit     set

/root:2018011..:wlan0:15s>
```

`help` followed by a command name gives details about that command (e.g., `> help batch`).

A.2.1 Parameters.

Getting a list of capture parameters and their values is performed with the `get` command:

```
/root:2018011..:wlan0:15s> get

Parameters:
bearing:       0
channel:       None
degrees:       360
duration:      15
focused:       None
hop_dist:      2
hop_int:       0.183296
iface:         wlan0
test:          20180118-03-53-15

Debug is False
HTTP server is False
/root:2018011..:wlan0:15s>
```

Capture parameters may be set with the `set` command, followed by the parameter and its new value (e.g., `> set channel 5`).

A.2.2 Debug Logging.

Writing debug messages to the console may be toggled with the `debug` command followed by a truth value, such as 1, True, On, etc (e.g., `> debug on`).

A.2.3 HTTP Server.

A HTTP server is available to serve up capture files easily by using the `serve` command followed by a truth value, such as 1, True, On, etc (e.g., `> serve 1`). This starts a HTTP server on port 80 in the current working directory of `localizer`, accessible from another computer at `http://<localizer-ip-address>`.

A.2.4 Wide Capture.

When an operator is ready to perform a wide capture, the `capture` command initiates the capture process described in Section A.3. Following the capture, all detected WAPs are displayed.

```
/root:2018011..:wlan0:15s> capture
Setting up threads                : 100%|XXXXXX| 4/4 [00:01<00:00,  3.00it/s]
Capturing packets for 15s         :  50%|XXXXXX| 8/16 [00:08<00:08,  1.00s/it
...

    ssid     bssid              channel     security    strength    method
1   <blank>  00:fe:c8:7d:ac:51  6           WPA         -19         pchip
2   <blank>  00:fe:c8:7d:ac:54  6           WPA         -21         pchip
3   <blank>  00:fe:c8:7d:ac:50  6           WPA         -21         pchip
4   <blank>  00:fe:c8:7d:ac:52  6           WPA         -21         pchip
...
```

A.2.5 Focused Capture.

Once a wide capture has been performed, the operator may perform a focused capture by using the `capture` command followed by a number corresponding to a WAP in the displayed results (e.g., `> capture 2`). After the wide capture is performed, the WAP table is updated with the new prediction.

A.2.6 Connect.

The operator may connect to a detected WAP with the `connect` command followed by a number corresponding to a WAP in the displayed results and followed by a password, if the WAP is password protected (e.g., `> connect 2 password123`). Once connected, `localizer` enters the `connect` shell. The only current command in the `connect` shell is `ping`, which attempts to perform a ping through the connected WAP.

A.3 Batch Capture

Batch captures are performed by entering batch mode from the interactive shell with the `batch` command. The primary difference of batch capture mode is that instead of interactively setting the capture parameters, parameters are imported from capture configuration files using the `import` command. The capture configurations used to capture experimental data are shown in Appendix D.

```
/root:2018011..:wlan0:15s> batch
You are now in batch processing mode. Type 'exit' to return
/root:batch> cd /capture
/capture:batch> import capture-3-test.conf
Found 1 batches
100%|XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX| 1/1 [00:00<00:00, 15.23it/s]
/capture:batch>
```

A.4 Batch Processing

Batch processing is performed from outside the `localizer` shells just discussed, with the `localizer` executable directly using the `-p` parameter.

```
$ localizer -p
Found 103 unprocessed data sets
```

```
Processing:   6%|XXX                              | 6/103 [00:19<05:16,  3.26s/it]
```

Batch processing recursively searches the provided directory, or current working directory if none was provided, for all directories with unprocessed capture data. When found, it spawns subprocesses to process all discovered unprocessed capture data in parallel.

Appendix B. `localizer` Source Code

B.1 Setup and Initialization Code

B.1.1 `setup.py`.

Python package `sextuplets` is used to install `localizer` on the system.

```
import sys

from setuptools import setup

if sys.version_info < (3,5):
    sys.exit('Sorry, Python < 3.5 is not supported')

def readme():
    with open('README.md') as f:
        return f.read()

setup(
    name='localizer',
    version='0.1',
    description="Signal Localizer: Data Gathering Tool for Radiolocation",
    long_description=readme(),
    url='https://github.com/elBradford/localizer',
    author='Bradford',
    packages=['localizer'],
    install_requires=[
        'pyshark',
        'gpsd-py3',
        'tqdm',
        'pandas',
        'scipy',
        'numexpr',
        'bottleneck',
        'numpy',
        'pigpio',
        'python-dateutil',
        'tabulate',
        'wifi==0.8.0rc1',
    ],
    test_suite='nose.collector',
    tests_require=['nose'],
    entry_points={
        'console_scripts': ['localizer=localizer.main:main'],
    },
    classifiers=[
        "Environment :: Console",
        "Operating System :: Unix",
        "Topic :: Scientific/Engineering",
```

```
        "Programming Language :: Python :: 3.5",
        "Intended Audience :: Science/Research"
    ],
    )
```

B.1.2 localizer/main.py.

This module serves as bootstrap code for the project. Argument parsing enables the different roles of the framework through passing different arguments from the command line.

```
import argparse
from os import getcwd

import localizer


# STARTUP
def main():

    parser = argparse.ArgumentParser()
    me_group = parser.add_mutually_exclusive_group()
    # TODO Implement command line capture and batch
    # group_capture = me_group.add_argument_group('Capture')
    # group_capture.add_argument("-c", "--capture")
    parser.add_argument("-d", "--debug",
                        help="Make debug output print to the console. This flag
                        ↪  may also be set in the shell",
                        action="store_true")
    parser.add_argument("-w", "--workingdir",
                        help="Set the parent directory for session experiments.
                        ↪  If blank, current directory is used.",
                        default=getcwd())
    me_group.add_argument("-p", "--process",
                          help="Process the files in the current directory, or a
                          ↪  provided working directory (-w)",
                          action="store_true")
    parser.add_argument("-m", "--macs",
                        help="If processing, a file containing mac addresses to
                        ↪  filter on")
    parser.add_argument("-ccw", "--counterclockwise",
                        help="Set this flag if the captures were performed in a
                        ↪  counter-clockwise direction",
                        action="store_true")
    me_group.add_argument("-s", "--shell",
                          help="Start the localizer shell",
                          action="store_true")
    parser.add_argument("--serve",
```

```
                        help="Serve files from the working directory on port 80.
                        ↪  This flag may also be set in the shell",
                        action="store_true")
    args = parser.parse_args()

    localizer.set_debug(args.debug)

    # Validate provided directory
    try:
        localizer.set_working_dir(args.workingdir)
    except ValueError as e:
        print(e)
        exit(1)

    if args.serve:
        localizer.set_serve(args.serve)

    if args.macs:
        args.macs = localizer.load_macs(args.macs)

    # Shell Mode
    if args.shell:
        from localizer.shell import LocalizerShell
        LocalizerShell(args.macs)

    elif args.process:
        from localizer import process
        process.process_directory(args.macs, not args.counterclockwise)

    elif args.serve:
        import socket
        input("Serving files from {} on {}:80, press any key to
        ↪  exit".format(getcwd(), socket.gethostname()))

    else:
        parser.print_help()


if __name__ == '__main__':
    main()
```

B.1.3 `localizer/__init__.py`.

This file contains initialization instructions for the `localizer` package, including package global variables, logging centralization, and various utilities.

This file also has code for enabling a simple http server, useful for pulling captured data from the prototype or DWAP.

```python
import atexit
import http.server
import logging
import os
import socketserver
from threading import Thread

from localizer.meta import Params

# Shared Variables
debug = False
serve = False

# Console colors
W = '\033[0m'  # white (normal)
R = '\033[31m'  # red
G = '\033[32m'  # green
O = '\033[33m'  # orange
B = '\033[34m'  # blue
P = '\033[35m'  # purple
C = '\033[36m'  # cyan
GR = '\033[37m'  # gray


# Set up logging
package_logger = logging.getLogger()
package_logger.setLevel(logging.DEBUG)
_console_handler = logging.StreamHandler()
_console_handler.setLevel(logging.WARNING)
_console_handler.setFormatter(logging.Formatter('%(name)s - %(levelname)s:
↪  %(message)s'))
package_logger.addHandler(_console_handler)

# Set up web server
PORT = 80
httpd = None
httpd_thread = None
socketserver.TCPServer.allow_reuse_address = True


def set_serve(value):
    global serve
    serve = value

    if serve:
        start_httpd()
    else:
        shutdown_httpd()


def restart_httpd():
    shutdown_httpd()
```

```python
    start_httpd()


def shutdown_httpd():
    global httpd, httpd_thread

    if httpd is not None:
        package_logger.info("Shutting down http server")
        httpd.shutdown()
        httpd = None
        httpd_thread.join()
        httpd_thread = None


def start_httpd():
    global httpd, httpd_thread
    if httpd is not None or httpd_thread is not None:
        shutdown_httpd()

    package_logger.info("Starting http server in {}".format(os.getcwd()))
    httpd = socketserver.TCPServer(("", PORT), QuietSimpleHTTPRequestHandler)
    httpd_thread = Thread(target=httpd.serve_forever)
    httpd_thread.daemon = True
    httpd_thread.start()


# Working Directory
_working_dir = None


def set_working_dir(path):
    global _working_dir

    if path == _working_dir:
        return

    _current_dir = os.getcwd()

    try:
        # cd into directory
        os.chdir(path)
        _new_path = os.getcwd()

        # Try to write and remove a tempfile to the directory
        _tmpfile = os.path.join(_new_path, 'tmpfile')
        with open(_tmpfile, 'w') as fp:
            fp.write(" ")
        os.remove(_tmpfile)

        # restart httpd if it's running
        if serve:
            restart_httpd()
```

```
            _working_dir = _new_path
    except (PermissionError, TypeError):
        os.chdir(_current_dir)
        raise ValueError("Cannot write to working directory '{}'".format(path))
    except FileNotFoundError:
        os.chdir(_current_dir)
        raise ValueError("Invalid directory '{}'".format(path))


# A quiet implementation of SimpleHTTPRequestHandler
class QuietSimpleHTTPRequestHandler(http.server.SimpleHTTPRequestHandler):
    def log_message(self, fmt, *args):
        pass


def set_debug(value):
    global debug, _console_handler
    debug = value
    if package_logger is not None:
        if debug:
            _console_handler.setLevel(logging.DEBUG)
        else:
            _console_handler.setLevel(logging.WARNING)

        package_logger.info("Debug set to {}".format(value))


def load_macs(mac_path):
    import csv
    with open(mac_path, 'r', newline='') as mac_tsv:
        csv_reader = csv.DictReader(mac_tsv, dialect="unix", delimiter='\t')
        return [line['BSSID'] for line in csv_reader]


# /dev/null, send output from programs so they don't print to screen.
DN = open(os.devnull, 'w')
ERRLOG = open(os.devnull, 'w')
OUTLOG = open(os.devnull, 'w')


@atexit.register
def cleanup():
    logging.shutdown()
```

B.2 Utilities

B.2.1 `localizer/meta.py`.

This module contains a class definition for the basic unit of a capture, a `Params` object. This object contains all the parameters needed for a capture to occur and enforces strict range and value requirements on each parameter. This module also contains miscellaneous meta-data and capture-related constants.

```python
import datetime
import re
import time

from geomag import WorldMagneticModel

import localizer

# WIFI Constants
IEEE80211bg = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
IEEE80211bg_intl = IEEE80211bg + [12, 13, 14]
IEEE80211a = [36, 40, 44, 48, 52, 56, 60, 64, 149, 153, 157, 161]
IEEE80211bga = IEEE80211bg + IEEE80211a
IEEE80211bga_intl = IEEE80211bg_intl + IEEE80211a
TU = 1024/1000000  # 1 TU = 1024 usec https://en.wikipedia.org/wiki/TU_(Time_Unit)
STD_BEACON_INT = 100*TU
OPTIMAL_BEACON_INT = 179*TU
STD_CHANNEL_DISTANCE = 2


meta_csv_fieldnames = ['name',
                       'pass',
                       'path',
                       'iface',
                       'duration',
                       'hop_int',
                       'pos_lat',
                       'pos_lon',
                       'pos_alt',
                       'pos_lat_err',
                       'pos_lon_err',
                       'pos_alt_err',
                       'start',
                       'end',
                       'degrees',
                       'bearing',
                       'pcap',
                       'nmea',
```

```
                       'coords',
                       'focused',
                       'guess',
                       'elapsed',
                       'num_guesses',
                       'guess_time',
                       ]


required_suffixes = {"nmea": ".nmea",
                    "pcap": ".pcapng",
                    "meta": "-capture.csv",
                    "coords": "-gps.csv",
                     }


capture_suffixes = {
                    "guess": "-guess.csv",
                    "results": "-results.csv",
                    "capture": "-capture.conf",
                    }

capture_suffixes.update(required_suffixes)


class Params:

    VALID_PARAMS = ["iface",
                    "duration",
                    "degrees",
                    "bearing",
                    "hop_int",
                    "hop_dist",
                    "mac",
                    "macs",
                    "channel",
                    "focused",
                    "capture"]

    def __init__(self,
                 iface=None,
                 duration=15.0,
                 degrees=360,
                 bearing=0,
                 hop_int=OPTIMAL_BEACON_INT,
                 hop_dist=STD_CHANNEL_DISTANCE,
                 macs=None,
                 channel=None,
                 focused=None,
                 capture=time.strftime('%Y%m%d-%H-%M-%S')):

        # Default Values
```

```
        self._duration = self._degrees = self._bearing = self._hop_int = self._hop_dist = self._macs = self._channel = self._focused = self._capture = None
        self._iface = iface
        self.duration = duration
        self.degrees = degrees
        self.bearing_magnetic = bearing
        self.hop_int = hop_int
        self.hop_dist = hop_dist
        self.macs = macs
        self.channel = channel
        self.focused = focused
        self.capture = capture

    @property
    def iface(self):
        return self._iface

    @iface.setter
    def iface(self, value):
        from localizer import interface
        if value in list(interface.get_interfaces()):
            self._iface = value
        else:
            raise ValueError("Invalid interface: {}".format(value))

    @property
    def duration(self):
        return self._duration

    @duration.setter
    def duration(self, value):
        try:
            if not isinstance(value, float):
                value = float(value)
            if value < 0:
                raise ValueError()
            self._duration = value
        except ValueError:
            raise ValueError("Invalid duration: {}; should be a float >= 0".format(value))

    @property
    def degrees(self):
        return self._degrees

    @degrees.setter
    def degrees(self, value):
        try:
            if not isinstance(value, int):
                value = int(float(value))
            self._degrees = value
```

```
        except ValueError as e:
            raise ValueError("Invalid degrees: {}; should be an int".format(value))

    @property
    def bearing_magnetic(self):
        return self._bearing

    @bearing_magnetic.setter
    def bearing_magnetic(self, value):
        try:
            if not isinstance(value, int):
                value = int(float(value))
            self._bearing = value % 360
        except ValueError:
            raise ValueError("Invalid bearing: {}; should be an int".format(value))

    def bearing_true(self, lat, lon, alt=0, date=datetime.date.today()):
        wmm = WorldMagneticModel()
        declination = wmm.calc_mag_field(lat, lon, alt, date).declination
        return self._bearing + declination

    @property
    def hop_int(self):
        return self._hop_int

    @hop_int.setter
    def hop_int(self, value):
        try:
            if not isinstance(value, float):
                value = round(float(value), 5)
            if value < 0:
                raise ValueError()
            self._hop_int = value
        except ValueError:
            raise ValueError("Invalid hop interval: {}; should be a float >= 0".format(value))

    @property
    def hop_dist(self):
        return self._hop_dist

    @hop_dist.setter
    def hop_dist(self, value):
        try:
            if not isinstance(value, int):
                value = int(value)
            if value <= 0:
                raise ValueError()
            self._hop_dist = value
        except ValueError:
```

```
            raise ValueError("Invalid hop distance: {}; should be an integer > 0".format(value))

    @property
    def macs(self):
        return self._macs

    @macs.setter
    def macs(self, value):
        self._macs = []
        if value:
            self.add_mac(value)

    def add_mac(self, value):
        try:
            # Check for string
            if isinstance(value, str):
                if self.validate_mac(value):
                    self._macs.append(value)
                else:
                    raise ValueError
            else:
                # Try to treat value as an iterable
                for mac in value:
                    if self.validate_mac(mac):
                        self._macs.append(mac)
                    else:
                        raise ValueError

        except (ValueError, TypeError):
            raise ValueError("Invalid mac address or list supplied; should be a mac string or list of mac strings")

    @property
    def channel(self):
        return self._channel

    @channel.setter
    def channel(self, value):
        try:
            if value is None:
                self._channel = value
            else:
                if not isinstance(value, int):
                    value = int(value)
                if value <= 0:
                    raise ValueError()
                self._channel = value
        except ValueError:
            raise ValueError("Invalid channel: {}; should be an integer > 0".format(value))

```

```python
    @property
    def focused(self):
        return self._focused

    @focused.setter
    def focused(self, value):
        try:
            if value is None:
                self._focused = value
            else:
                if not isinstance(value, tuple) or len(value) != 2:
                    raise ValueError()
                else:
                    _degrees = float(value[0])
                    if _degrees <= 0 or _degrees > 360:
                        raise ValueError()
                    _duration = float(value[1])
                    if _duration <= 0:
                        raise ValueError()

                    self._focused = (_degrees, _duration)
        except ValueError:
            raise ValueError("Invalid fine: {}; should be a tuple of length 2 (degrees[width], duration > 0)".format(value))

    @property
    def capture(self):
        return self._capture

    @capture.setter
    def capture(self, value):
        self._capture = str(value)

    # Validation functions
    def validate_antenna(self):
        return self.duration is not None and \
               self.degrees is not None and \
               self.bearing_magnetic is not None

    def validate_gps(self):
        return self.duration is not None

    def validate_capture(self):
        return self.iface is not None and \
               self.duration is not None

    def validate_wifi(self):
        return self.iface is not None and \
               self.duration is not None and \
               self.hop_int is not None

    @staticmethod
```

```python
    def validate_mac(mac):
        return re.match("[0-9a-f]{2}([-:])[0-9a-f]{2}(\\1[0-9a-f]{2}){4}$", mac.lower())

    def validate(self):
        return self.validate_antenna() and self.validate_gps() and self.validate_wifi()

    def __str__(self):
        retstr = "\n{} \tParameters: {}\n".format(localizer.G, localizer.W)
        for param, val in sorted(self.__dict__.items()):

            # If no macs are specified, don't print
            if param is '_macs':
                if len(val) > 0:
                    retstr += "\t    Macs:\n"
                    for i, mac in enumerate(val):
                        retstr += "\t\t    {:<15}{:<15}\n".format(i, mac)
            else:
                # Highlight 'None' values as red, except for 'test' which is optional
                signifier = ''
                if param is not '_capture' and val is None:
                    signifier = localizer.R
                retstr += "\t    {:<15}{}{:<15}{}\n".format(str(param[1:]) + ': ', signifier, str(val), localizer.W)

        return retstr

    def copy(self):
        from copy import deepcopy

        return Params(
            self.iface,
            self.duration,
            self.degrees,
            self.bearing_magnetic,
            self.hop_int,
            self.hop_dist,
            deepcopy(self.macs),
            self.channel,
            deepcopy(self.focused),
            self.capture
        )
```

B.2.2 `localizer/locate.py`.

The code in this module provides the interpolation methods described in chapter V - this enables the software to take any number of captured beacons and make an optimal guess as to the likely location of the emitter.

```python
import numpy as np
import pandas as pd


def locate_naive(series):
    if len(series) > 360:
        series = series[np.arange(0, 360)]

    return series.idxmax()


def locate_interpolate(series_concat, method):
    series_inter = series_concat.interpolate(method=method)[np.arange(0, 360)]

    return series_inter.idxmax()


def prep_for_interpolation(dataframe, bearing, x='bearing_magnetic', y='mw'):
    """
    Prepare a dataframe for interpolation by stripping extraneous columns and converting it into a series
    """

    # Stip columns and convert to series
    df = dataframe.filter([x, y]).rename(columns={x: 'deg'}).sort_values('deg')
    df['deg'] = np.round(df['deg'])

    if df.duplicated('deg', keep=False).any():
        df = df.groupby('deg', group_keys=False).apply(lambda z: z.loc[z.mw.idxmax()])

    series_mid = df.set_index('deg').reindex(np.arange(0, 360)).iloc[:, 0]

    if bearing >= 360:
        # Extend to the left and right in order to ease interpolation
        series_left = series_mid.copy()
        series_left.index = np.arange(-360, 0)
        series_right = series_mid.copy()
        series_right.index = np.arange(360, 720)

        series_concat = pd.concat([series_left, series_mid, series_right])

        return series_concat
    else:
```

```
        return series_mid


def interpolate(series, bearing):
    """
    Interpolate the given series in the best manner based on testing
    :param series: Pandas Series
    :param expand_to_360: Whether to expand series so that it properly wraps around 360 degrees
    :return:
    """

    if 0 > len(series) <= 1:
        _method = 'slinear'
    elif 1 > len(series) <= 2:
        _method = 'naive'
    else:
        _method = 'pchip'

    _guess = _error_methods[_method](prep_for_interpolation(series, bearing))
    return _guess, _method


_error_methods = {
    'naive': locate_naive,
    'quadratic': lambda series: locate_interpolate(series, 'quadratic'),
    'cubic': lambda series: locate_interpolate(series, 'cubic'),
    'linear': lambda series: locate_interpolate(series, 'linear'),
    'slinear': lambda series: locate_interpolate(series, 'slinear'),
    'barycentric': lambda series: locate_interpolate(series, 'barycentric'),
    'krogh': lambda series: locate_interpolate(series, 'krogh'),
    'piecewise_polynomial': lambda series: locate_interpolate(series, 'piecewise_polynomial'),
    'from_derivatives': lambda series: locate_interpolate(series, 'from_derivatives'),
    'pchip': lambda series: locate_interpolate(series, 'pchip'),
    'akima': lambda series: locate_interpolate(series, 'akima'),
}
```

B.2.3 `localizer/shell.py`.

This module manages the shell, which provides the interactive shell and batch capture roles. Multiple subclasses of the `Cmd` class provide the necessary features for these roles.

```
import abc
import configparser
import csv
```

```python
import datetime
import logging
import os
import pprint
import subprocess
import time
from cmd import Cmd
from distutils.util import strtobool

from tqdm import tqdm

import localizer
from localizer import capture, process, meta, antenna, interface
from localizer.capture import APs

module_logger = logging.getLogger(__name__)
_file_handler = logging.FileHandler('localizer.log')
_file_handler.setLevel(logging.DEBUG)
_file_handler.setFormatter(logging.Formatter('%(asctime)s - %(name)s -
↪  %(levelname)s: %(message)s'))
module_logger.addHandler(_file_handler)
module_logger.info("****STARTING LOCALIZER****")


# Helper class for exit functionality
class ExitCmd(Cmd):
    @staticmethod
    def can_exit():
        return True

    def onecmd(self, line):
        r = super().onecmd(line)
        if r and (self.can_exit() or input('exit anyway ? (yes/no):') == 'yes'):
            return True
        return False

    @staticmethod
    def do_exit(_):
        """Exit the interpreter."""
        return True

    @staticmethod
    def do_quit(_):
        """Exit the interpreter."""
        return True

    def emptyline(self):
        pass


# Helper class for shell command functionality
class ShellCmd(Cmd, object):
```

```python
    @staticmethod
    def do_shell(args):
        """Execute shell commands in the format 'shell <command>'"""
        os.system(args)


# Helper class for debug toggling
class DebugCmd(Cmd, object):

    @staticmethod
    def do_debug(args):
        """
        Sets printing of debug information or shows current debug level if no param given

        :param args: (Optional) Set new debug value.
        :type args: str
        """

        args = args.split()
        if len(args) > 0:
            try:
                val = strtobool(args[0])
                localizer.set_debug(val)
            except ValueError:
                module_logger.error("Could not understand debug value '{}'".format(args[0]))

        print("Debug is {}".format("ENABLED" if localizer.debug else "DISABLED"))


# Helper class for cd and directory functions
class DirCmd(Cmd, object, metaclass=abc.ABCMeta):

    def do_cd(self, args):
        """
        cd into specified path

        :param args: path to cd into
        """

        args = args.split()
        if len(args) == 0:
            print(os.getcwd())
        else:
            try:
                localizer.set_working_dir(args[0])

            except ValueError as e:
                module_logger.error(e)
            finally:
                self._update_prompt()
```

```python

    @abc.abstractmethod
    def _update_prompt(self):
        raise NotImplementedError("Subclasses of this class must implement _update_prompt")


# Base Localizer Shell Class
class LocalizerShell(ExitCmd, ShellCmd, DirCmd, DebugCmd):

    def __init__(self, macs=None):
        super().__init__()

        self._modules = ["antenna", "gps", "capture", "wifi"]
        self._params = meta.Params()
        if macs:
            self._params.macs = macs
        self._aps = APs()

        # Ensure we have root
        if os.getuid() != 0:
            print("Error: this application needs root to run correctly. Please run as root.")
            exit(1)

        # WiFi
        module_logger.info("Initializing WiFi")
        # Set interface to first
        iface = interface.get_first_interface()
        if iface is not None:
            self._params.iface = iface
        else:
            module_logger.error("No valid wireless interface available")
            exit(1)

        # Start the command loop - these need to be the last lines in the initializer
        self._update_prompt()
        self.cmdloop('Welcome to Localizer Shell...')

    @staticmethod
    def do_serve(args):
        """
        Sets serving of the working directory over http:80, or shows current setting if no param given

        :param args: (Optional) Set new serve value.
        :type args: str
        """

        args = args.split()
        if len(args) > 0:
```

```
            try:
                val = strtobool(args[0])
                localizer.set_serve(val)
            except ValueError:
                module_logger.error("Could not understand serve value '{}'".format(args[0]))

        print("Serve is {}".format("ENABLED" if localizer.serve else "DISABLED"))
        if localizer.serve:
            print("HTTP serving working dir {} on port :{}".format(os.getcwd(), localizer.PORT))

    @staticmethod
    def do_process(_):
        """
        Process the results of all captures in the current working directory.
        This command will look in each subdirectory of the current path for unprocessed captures
        It looks for valid *-capture.csv, etc, and processes the files to build *.results.csv
        """

        _processed = process.process_directory()

        print("Processed {} captures".format(_processed))

    def do_set(self, args):
        """
        Set a named parameter. All parameters require a value except for iface and macs
        - iface without a parameter will set the iface to the first system wireless iface found
        - macs without a parameter will delete the mac address whitelist

        :param args: Parameter name followed by new value
        :type args: str
        """

        split_args = args.split()
        if len(split_args) < 1:
            module_logger.error("You must provide at least one argument".format(args))
        elif len(split_args) == 1:
            if split_args[0] == "iface":
                iface = interface.get_first_interface()

                if iface is not None:
                    self._params.iface = iface
                else:
                    module_logger.error("There are no wireless interfaces available.")
            elif split_args[0] == 'macs':
```

```
                self._params.macs = []
            else:
                module_logger.error("Parameters require a value".format(split_args[0]))
        elif split_args[0] in meta.Params.VALID_PARAMS:
            try:
                param = split_args[0]
                value = split_args[1]
                # Validate certain parameters
                if split_args[0] == "iface":
                    self._params.iface = value
                elif param == "duration":
                    self._params.duration = value
                elif param == "degrees":
                    self._params.degrees = value
                elif param == "bearing":
                    self._params.bearing_magnetic = value
                elif param == "hop_int":
                    self._params.hop_int = value
                elif param == "hop_dist":
                    self._params.hop_dist = value
                elif param == "mac":
                    self._params.add_mac(value)
                elif param == "macs":
                    # Load macs from provided file
                    self._params.add_mac(localizer.load_macs(value))
                elif param == "channel":
                    self._params.channel = value
                elif param == "capture":
                    self._params.capture = value

                print("Parameter '{}' set to '{}'".format(param, value))

            except (ValueError, FileNotFoundError) as e:
                module_logger.error(e)
        else:
            module_logger.error("Invalid parameter '{}'".format(split_args[0]))

        self._update_prompt()

    def do_get(self, args):
        """
        View the specified parameter or all parameters if none specified. May also view system interface data

        :param args: param name, ifaces for system interfaces, or blank for all parameters
        :type args: str
        """

        split_args = args.split()

```

```python
        if len(split_args) >= 1:
            if split_args[0] == "ifaces":
                pprint.pprint(interface.get_interfaces())
            elif split_args[0] == "params":
                print(str(self._params))
            elif split_args[0] == "bearing":
                print("Current bearing: {} degrees".format(antenna.bearing_current))
            else:
                module_logger.error("Unknown parameter '{}'".format(split_args[0]))
        else:
            pprint.pprint(interface.get_interfaces())
            print(str(self._params))
            print("Debug is {}".format(localizer.debug))
            print("HTTP server is {}".format(localizer.serve))

    def do_list(self, _):
        """
        List any detected access points, their bearing, and whether they have been scanned
        """

        if self._aps:
            print(self._aps)
        else:
            print("No detected aps, or scan hasn't been performed")

    def do_capture(self, args):
        """
        Start the capture with the needed parameters set
        """

        split_args = args.split()

        if len(split_args) >= 1 and int(split_args[0]) < len(self._aps):
            # Build focused capture based on selected access point
            _ap = self._aps[int(split_args[0])]
            _prediction = _ap.bearing
            _bearing = _prediction - capture.OPTIMAL_CAPTURE_DEGREES_FOCUSED/2
            _duration = antenna.FOCUSED_RATE[capture.OPTIMAL_CAPTURE_DEGREES_FOCUSED] * capture.OPTIMAL_CAPTURE_DEGREES_FOCUSED / 360
            _channel = _ap.channel
            _bssid = _ap.bssid
            _try_params = localizer.meta.Params(self._params.iface, _duration, capture.OPTIMAL_CAPTURE_DEGREES_FOCUSED, _bearing, hop_int=0, channel= _channel, macs=[_bssid])
            module_logger.info("Setting capture to focused mode")
        else:
            _try_params = self._params

```

```
        if not _try_params.validate():
            module_logger.error("You must set 'iface' and 'duration' parameters first")
        else:
            # Shutdown http server if it's on
            localizer.shutdown_httpd()

            module_logger.info("Starting capture")
            try:
                _result = capture.capture(_try_params, reset=_try_params.bearing_magnetic)
                if _result:
                    _capture_path, _meta = _result

                    with open(os.path.join(_capture_path, _meta), 'rt') as meta_csv:
                        _meta_reader = csv.DictReader(meta_csv, dialect='unix')
                        meta = next(_meta_reader)

                    _, _, _, _aps = process.process_capture(meta, _capture_path, write_to_disk=False, guess=True, macs=_try_params.macs)
                    if len(self._aps):
                        self._aps.update(_aps)
                    else:
                        self._aps.aps = _aps
                    print(self._aps)
                else:
                    raise RuntimeError("Capture failed")

            except RuntimeError as e:
                module_logger.error(e)

            finally:
                # Restart http server if it is supposed to be on
                if localizer.serve:
                    localizer.start_httpd()

    def do_connect(self, args):
        """
        Connect to the specified access point number from the list command with the provided password.
        """
        split_args = args.split()

        if len(split_args) >= 2 and int(split_args[0]) < len(self._aps):
            # Build focused capture based on selected access point
            _ap = self._aps[int(split_args[0])]
            _prediction = int(_ap.bearing)
            # Set antenna to predicted bearing
            antenna.AntennaThread.reset_antenna(_prediction)

            # Connect to the access point
```

```
            try:
                WiFiConnectShell(self._params.iface, _ap.ssid, split_args[1])
            except ValueError as e:
                module_logger.error(e)
        else:
            print("You must provide an AP number and a password")

    @staticmethod
    def do_batch(_):
        """
        Start batch mode
        """

        BatchShell()

    def _update_prompt(self):
        """
        Update the command prompt based on the iface and duration parameters
        """

        elements = [localizer.GR + os.getcwd()]
        if self._params.capture:
            capture = (self._params.capture[:7] + '..') if len(self._params.capture) > 9 else self._params.capture
            elements.append(localizer.G + capture)
        if self._params.iface is not None:
            elements.append(localizer.C + self._params.iface)
        if self._params.duration > 0:
            elements.append(localizer.GR + str(self._params.duration) + 's')

        separator = localizer.W + ':'
        self.prompt = separator.join(elements) + localizer.W + '> '


class BatchShell(ExitCmd, ShellCmd, DirCmd, DebugCmd):

    def __init__(self):
        super().__init__()

        self._pause = True
        self._batches = []

        # Start the command loop - these need to be the last lines in the initializer
        self._update_prompt()
        self.cmdloop("You are now in batch processing mode. Type 'exit' to return to the capture shell")

    def do_import(self, args):
        """
        Import all captures in the current directory, or the capture name provided. Captures are files that end in -capture.conf
```

```
        :param args: capture to import
        :type args: str
        """

        _filenames = []

        args = args.split()
        # Check for provided filename
        if len(args):
            for arg in args:
                if os.path.isfile(arg):
                    _filenames.append(arg)
                else:
                    if os.path.isfile(arg + meta.capture_suffixes['capture']):
                        _filenames.append(arg + meta.capture_suffixes['capture'])
        else:
            # Get list of valid capture batches in current directory
            _filenames = [file for file in next(os.walk('.'))[2] if
            ↪ file.endswith(meta.capture_suffixes['capture'])]

        print("Found {} batches".format(len(_filenames)))

        # Import captures from each batch
        _count = 0
        for batch in tqdm(_filenames):
            try:
                _name, _passes, _captures = BatchShell._parse_batch(batch)
                self._batches.append((_name, _passes, _captures))
                _count += 1
            except ValueError as e:
                module_logger.error(e)

        logging.info("Imported {} batches".format(_count))

    def complete_import(self, text, _, __, ___):
        return [file for file in next(os.walk('.'))[2] if file.startswith(text)
        ↪ and file.endswith(meta.capture_suffixes['capture'])]

    def do_capture(self, _):
        """
        Run all the imported captures
        """

        if not self._batches:
            print("No batches have been imported")
        else:
            _total = 0
            for _, _passes, _captures in self._batches:
                _total += len(_captures)*_passes

            _start_time = time.time()
```

```python
            print("Starting batch of {} captures".format(_total))
            _curr = 0
            for _, _passes, _captures in self._batches:
                _len_pass = len(str(_passes))
                for cap in _captures:
                    for p in range(_passes):
                        print(localizer.R + "Capture {:>4}/{}\t\t{} elapsed".format(_curr, _total, datetime.timedelta(seconds=time.time()-_start_time)) + localizer.W)
                        capture.capture(cap, str(p).zfill(_len_pass), cap.bearing_magnetic)
                        _curr += 1

            print("Complete - total time elapsed: {}".format(datetime.timedelta(seconds=time.time()-_start_time)))

    def do_get(self, _):
        """
        Print the captures
        """

        for _name, _passes, _captures in self._batches:
            for cap in _captures:
                print(cap)

            print("Batch: {}; {} captures, {} passes each".format(_name, len(_captures), _passes))

        print("Estimated total runtime: {:0>8}".format(str(self._calculate_runtime())))

    def do_pause(self, args):
        """
        Pause between captures to allow for antenna calibration

        :param args: True to pause between captures, False to continue to the next capture immediately
        :type args: str
        """

        args = args.split()
        if len(args) > 0:
            try:
                self._pause = strtobool(args[0])
            except ValueError:
                module_logger.error("Could not understand pause value '{}'".format(args[0]))

        print("Pause is {}".format("ENABLED" if self._pause else "DISABLED"))

    def do_clear(self, _):
```

```
        """
        Clear all batches
        """

        self._batches = []

    def _calculate_runtime(self):
        """
        Calculate an estimated runtime for the imported captures
        :return: Estimated runtime
        :rtype: int
        """

        _time = 0
        for _, _passes, _captures in self._batches:
            for cap in _captures:
                _time_temp = ((cap.duration * _passes))

                if cap.focused:
                    _nmacs = len(cap.macs)
                    _deg, _dur = cap.focused
                    _time_fine = (_deg * _dur) / 360
                    _time_fine *= _nmacs
                    _time_temp += _time_fine

                _time += _time_temp

        return datetime.timedelta(seconds=_time)


    @staticmethod
    def _parse_batch(file):
        """
        Import captures from the supplied batch file

        :param file: Path to the file to import
        :type file: str
        :return: A tuple containing a passes value and a list containing captures
        :rtype: (str, int, list)
        """

        _name = file[:file.find(meta.capture_suffixes['capture'])]
        _captures = []

        config = configparser.ConfigParser()
        config.read(file, encoding='ascii')

        if not len(config.sections()):
            raise ValueError("Invalid capture config file: {}".format(file))

        _passes = int(config['meta']['passes'])

```

```
        for section in config.sections():
            if section == 'meta':
                continue

            cap = BatchShell._build_capture(config[section], config['meta'])
            if cap:
                _captures.append(cap)

        print("Imported {}/{} captures from {} batch ({} passes)".format(len(_captures), len(config.sections()) - 1, _name, _passes))
        return _name, _passes, _captures

    @staticmethod
    def _build_capture(capture_section, meta_section):
        """
        Use a dictionary from configparser to build a capture object

        :param capture_section: A dictionary of key and values with capture properties
        :type capture_section: dict
        :param meta_section: A dictionary of key and values with default properties
        :type meta_section: dict
        :return: A Params object
        :rtype: Params()
        """

        try:
            if 'iface' in capture_section and capture_section['iface']:
                _iface = capture_section['iface']
            elif 'iface' in meta_section and meta_section['iface']:
                _iface = meta_section['iface']
            else:
                _iface = interface.get_first_interface()
                if not _iface:
                    raise ValueError("No valid interface provided or available on system")

            if 'duration' in capture_section:
                _duration = capture_section['duration']
            elif 'duration' in meta_section:
                _duration = meta_section['duration']
            else:
                _duration = capture.OPTIMAL_CAPTURE_DURATION

            if 'degrees' in capture_section:
                _degrees = capture_section['degrees']
            elif 'degrees' in meta_section:
                _degrees = meta_section['degrees']
            else:
                raise ValueError("No valid degrees")
```

```
        if 'bearing' in capture_section:
            _bearing = capture_section['bearing']
        elif 'bearing' in meta_section:
            _bearing = meta_section['bearing']
        else:
            raise ValueError("No valid bearing")

        if 'hop_int' in capture_section:
            _hop_int = capture_section['hop_int']
        elif 'hop_int' in meta_section:
            _hop_int = meta_section['hop_int']
        else:
            _hop_int = interface.OPTIMAL_BEACON_INT

        if 'hop_dist' in capture_section:
            _hop_dist = capture_section['hop_dist']
        elif 'hop_dist' in meta_section:
            _hop_dist = meta_section['hop_dist']
        else:
            _hop_dist = interface.STD_CHANNEL_DISTANCE

        if 'capture' in capture_section:
            _capture = capture_section['capture']
        elif 'capture' in meta_section:
            _capture = meta_section['capture']
        else:
            raise ValueError("No valid capture name")

        if 'macs' in capture_section:
            _macs = capture_section['macs'].split(',')
        elif 'macs' in meta_section:
            _macs = meta_section['macs'].split(',')
        else:
            _macs = None

        if 'channel' in capture_section:
            _channel = capture_section['channel']
        elif 'channel' in meta_section:
            _channel = meta_section['channel']
        else:
            _channel = None

        if 'focused' in capture_section:
            _focused = tuple(capture_section['focused'].split(','))
        elif 'focused' in meta_section:
            _focused = tuple(meta_section['focused'].split(','))
        else:
            _focused = None

        cap = localizer.meta.Params(_iface, _duration, _degrees, _bearing,
        ↪  _hop_int, _hop_dist, _macs, _channel, _focused, _capture)
```

```python
            # Validate iface
            module_logger.debug("Setting iface {}".format(_iface))
            cap.iface = _iface

            return cap

        except ValueError as e:
            module_logger.warning(e)
            return None

    def _update_prompt(self):
        self.prompt = localizer.GR + os.getcwd() + localizer.W + ":" + localizer.G + "batch" + localizer.W + "> "


class WiFiConnectShell(ExitCmd, ShellCmd, DirCmd, DebugCmd):
    connect_timeout = 5

    def __init__(self, iface, ap, password):
        super().__init__()

        self._iface = iface
        self._ap = ap
        self._pw = password

        # Kill any existing wpa_supplicant instance
        subprocess.run(['killall', 'wpa_supplicant'])

        self._mode = interface.get_interface_mode(self._iface)
        # Take interface out of monitor mode
        if self._mode != "managed":
            interface.set_interface_mode(self._iface, "managed")

        print("Connecting to {}...".format(self._ap))
        # Try to connect - timeout if otherwise
        self._proc = subprocess.Popen(['/bin/bash', '-c', 'wpa_supplicant -i {} -c <(wpa_passphrase {} {})'.format(self._iface, self._ap, self._pw)], stdout=subprocess.PIPE, stderr=subprocess.PIPE)
        # Wait for process to output "File: ..." to stderr and then set flag for other threads
        curr_line = ""
        try:
            _time_waited = 0
            while "CTRL-EVENT-CONNECTED" not in curr_line:
                curr_line = self._proc.stdout.readline().decode()
                module_logger.debug("wpa_supplicant: {}".format(curr_line))
                time.sleep(.1)
                _time_waited += .1
                if _time_waited >= WiFiConnectShell.connect_timeout:
                    raise TimeoutError()
        except TimeoutError:
            self.do_disconnect(None)
```

```python
            raise ValueError("Timed out connecting to {}".format(self._ap))

        print("Getting IP address, waiting 10 seconds...")
        try:
            subprocess.run(['dhclient', self._iface], timeout=10)
        except subprocess.TimeoutExpired:
            self.do_disconnect(None)
            raise ValueError("Timed out getting IP address")

        # Start the command loop - these need to be the last lines in the initializer
        self._update_prompt()
        self.cmdloop("You are now connected to {}. Type 'disconnect' to disconnect and return to the capture shell".format(self._ap))

    def do_ping(self, args):
        """
        Send a ping request to 8.8.8.8 or a provided IP to check internet connectivity
        """
        _ip = "8.8.8.8"

        arg_split = args.split()
        if len(arg_split) > 0:
            _ip = arg_split[0]

        with subprocess.Popen(['ping', _ip, '-c', str(3), '-I', self._iface], stdout=subprocess.PIPE, bufsize=1, universal_newlines=True) as _proc:
            for line in _proc.stdout:
                print(line, end='')

    def do_disconnect(self, _):
        """
        Disconnect from the current AP
        """
        self._proc.kill()
        subprocess.run(['killall', 'wpa_supplicant'])
        interface.set_interface_mode(self._iface, self._mode)
        return self.do_exit(None)

    def _update_prompt(self):
        self.prompt = localizer.GR + os.getcwd() + localizer.W + ":" + localizer.G + "connect:" + self._ap + localizer.W + "> "
```

B.3 Capture & Processing

B.3.1 `localizer/capture.py`.

This module is orchestrates all the capture-related modules, such as `localizer/antenna.py`, `localizer/gps.py`, and `localizer/interface.py`, found in sections B.3.2, B.3.3, and B.3.4 respectively.

The function of this code is described in detail in 3.4.2.

```
import datetime
import re
import time

from geomag import WorldMagneticModel

import localizer

# WIFI Constants
IEEE80211bg = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
IEEE80211bg_intl = IEEE80211bg + [12, 13, 14]
IEEE80211a = [36, 40, 44, 48, 52, 56, 60, 64, 149, 153, 157, 161]
IEEE80211bga = IEEE80211bg + IEEE80211a
IEEE80211bga_intl = IEEE80211bg_intl + IEEE80211a
TU = 1024/1000000  # 1 TU = 1024 usec
↪  https://en.wikipedia.org/wiki/TU_(Time_Unit)
STD_BEACON_INT = 100*TU
OPTIMAL_BEACON_INT = 179*TU
STD_CHANNEL_DISTANCE = 2


meta_csv_fieldnames = ['name',
                       'pass',
                       'path',
                       'iface',
                       'duration',
                       'hop_int',
                       'pos_lat',
                       'pos_lon',
                       'pos_alt',
                       'pos_lat_err',
                       'pos_lon_err',
                       'pos_alt_err',
                       'start',
                       'end',
                       'degrees',
                       'bearing',
                       'pcap',
                       'nmea',
```

```
                        'coords',
                        'focused',
                        'guess',
                        'elapsed',
                        'num_guesses',
                        'guess_time',
                        ]


required_suffixes = {"nmea": ".nmea",
                     "pcap": ".pcapng",
                     "meta": "-capture.csv",
                     "coords": "-gps.csv",
                      }


capture_suffixes = {
                     "guess": "-guess.csv",
                     "results": "-results.csv",
                     "capture": "-capture.conf",
                     }

capture_suffixes.update(required_suffixes)


class Params:

    VALID_PARAMS = ["iface",
                     "duration",
                     "degrees",
                     "bearing",
                     "hop_int",
                     "hop_dist",
                     "mac",
                     "macs",
                     "channel",
                     "focused",
                     "capture"]

    def __init__(self,
                 iface=None,
                 duration=15.0,
                 degrees=360,
                 bearing=0,
                 hop_int=OPTIMAL_BEACON_INT,
                 hop_dist=STD_CHANNEL_DISTANCE,
                 macs=None,
                 channel=None,
                 focused=None,
                 capture=time.strftime('%Y%m%d-%H-%M-%S')):

        # Default Values
```

```
        self._duration = self._degrees = self._bearing = self._hop_int = self._hop_dist = self._macs = self._channel = self._focused = self._capture = None
        self._iface = iface
        self.duration = duration
        self.degrees = degrees
        self.bearing_magnetic = bearing
        self.hop_int = hop_int
        self.hop_dist = hop_dist
        self.macs = macs
        self.channel = channel
        self.focused = focused
        self.capture = capture

    @property
    def iface(self):
        return self._iface

    @iface.setter
    def iface(self, value):
        from localizer import interface
        if value in list(interface.get_interfaces()):
            self._iface = value
        else:
            raise ValueError("Invalid interface: {}".format(value))

    @property
    def duration(self):
        return self._duration

    @duration.setter
    def duration(self, value):
        try:
            if not isinstance(value, float):
                value = float(value)
            if value < 0:
                raise ValueError()
            self._duration = value
        except ValueError:
            raise ValueError("Invalid duration: {}; should be a float >= 0".format(value))

    @property
    def degrees(self):
        return self._degrees

    @degrees.setter
    def degrees(self, value):
        try:
            if not isinstance(value, int):
                value = int(float(value))
            self._degrees = value
```

```
        except ValueError as e:
            raise ValueError("Invalid degrees: {}; should be an int".format(value))

    @property
    def bearing_magnetic(self):
        return self._bearing

    @bearing_magnetic.setter
    def bearing_magnetic(self, value):
        try:
            if not isinstance(value, int):
                value = int(float(value))
            self._bearing = value % 360
        except ValueError:
            raise ValueError("Invalid bearing: {}; should be an int".format(value))

    def bearing_true(self, lat, lon, alt=0, date=datetime.date.today()):
        wmm = WorldMagneticModel()
        declination = wmm.calc_mag_field(lat, lon, alt, date).declination
        return self._bearing + declination

    @property
    def hop_int(self):
        return self._hop_int

    @hop_int.setter
    def hop_int(self, value):
        try:
            if not isinstance(value, float):
                value = round(float(value), 5)
            if value < 0:
                raise ValueError()
            self._hop_int = value
        except ValueError:
            raise ValueError("Invalid hop interval: {}; should be a float >= 0".format(value))

    @property
    def hop_dist(self):
        return self._hop_dist

    @hop_dist.setter
    def hop_dist(self, value):
        try:
            if not isinstance(value, int):
                value = int(value)
            if value <= 0:
                raise ValueError()
            self._hop_dist = value
        except ValueError:
```

```python
            raise ValueError("Invalid hop distance: {}; should be an integer > 0".format(value))

    @property
    def macs(self):
        return self._macs

    @macs.setter
    def macs(self, value):
        self._macs = []
        if value:
            self.add_mac(value)

    def add_mac(self, value):
        try:
            # Check for string
            if isinstance(value, str):
                if self.validate_mac(value):
                    self._macs.append(value)
                else:
                    raise ValueError
            else:
                # Try to treat value as an iterable
                for mac in value:
                    if self.validate_mac(mac):
                        self._macs.append(mac)
                    else:
                        raise ValueError

        except (ValueError, TypeError):
            raise ValueError("Invalid mac address or list supplied; should be a mac string or list of mac strings")

    @property
    def channel(self):
        return self._channel

    @channel.setter
    def channel(self, value):
        try:
            if value is None:
                self._channel = value
            else:
                if not isinstance(value, int):
                    value = int(value)
                if value <= 0:
                    raise ValueError()
                self._channel = value
        except ValueError:
            raise ValueError("Invalid channel: {}; should be an integer > 0".format(value))

```

```
    @property
    def focused(self):
        return self._focused

    @focused.setter
    def focused(self, value):
        try:
            if value is None:
                self._focused = value
            else:
                if not isinstance(value, tuple) or len(value) != 2:
                    raise ValueError()
                else:
                    _degrees = float(value[0])
                    if _degrees <= 0 or _degrees > 360:
                        raise ValueError()
                    _duration = float(value[1])
                    if _duration <= 0:
                        raise ValueError()

                    self._focused = (_degrees, _duration)
        except ValueError:
            raise ValueError("Invalid fine: {}; should be a tuple of length 2
            ↪  (degrees[width], duration > 0)".format(value))

    @property
    def capture(self):
        return self._capture

    @capture.setter
    def capture(self, value):
        self._capture = str(value)

    # Validation functions
    def validate_antenna(self):
        return self.duration is not None and \
               self.degrees is not None and \
               self.bearing_magnetic is not None

    def validate_gps(self):
        return self.duration is not None

    def validate_capture(self):
        return self.iface is not None and \
               self.duration is not None

    def validate_wifi(self):
        return self.iface is not None and \
               self.duration is not None and \
               self.hop_int is not None

    @staticmethod
```

```
    def validate_mac(mac):
        return re.match("[0-9a-f]{2}([-:])[0-9a-f]{2}(\\1[0-9a-f]{2}){4}$",
        ↪  mac.lower())

    def validate(self):
        return self.validate_antenna() and self.validate_gps() and
        ↪  self.validate_wifi()

    def __str__(self):
        retstr = "\n{} \tParameters: {}\n".format(localizer.G, localizer.W)
        for param, val in sorted(self.__dict__.items()):

            # If no macs are specified, don't print
            if param is '_macs':
                if len(val) > 0:
                    retstr += "\t    Macs:\n"
                    for i, mac in enumerate(val):
                        retstr += "\t\t    {:<15}{:<15}\n".format(i, mac)
            else:
                # Highlight 'None' values as red, except for 'test' which is
                ↪  optional
                signifier = ''
                if param is not '_capture' and val is None:
                    signifier = localizer.R
                retstr += "\t    {:<15}{}{:<15}{}\n".format(str(param[1:]) + ':
                ↪  ', signifier, str(val), localizer.W)

        return retstr

    def copy(self):
        from copy import deepcopy

        return Params(
            self.iface,
            self.duration,
            self.degrees,
            self.bearing_magnetic,
            self.hop_int,
            self.hop_dist,
            deepcopy(self.macs),
            self.channel,
            deepcopy(self.focused),
            self.capture
        )
```

B.3.2 localizer/antenna.py.

```
import atexit
import logging
import math
import threading
```

```python
import time
from subprocess import run

import pigpio

module_logger = logging.getLogger(__name__)

# Always start due north (magnetic) or change this variable
bearing_default = 0
bearing_current = bearing_default
bearing_max = 720
bearing_min = -360

# Constants
# Reset Rate Curve
# From https://mycurvefit.com/
#           0                  20
#          90                   7
#         180                   5
#         360                   4
get_reset_rate = lambda x: 3.235294 + (20 - 3.235294) / (1 + (x / 34.68111) ** 1.29956)
RESET_RATE = [get_reset_rate(x) for x in range(1080)]
get_focused_rate = lambda x: -4 + (20 + 4) / (1 + (x / 180) ** 0.48542683)
FOCUSED_RATE = [get_focused_rate(x) for x in range(360)]
#-4.           20.          180.            0.48542683

# Default number of steps per radian
steps_per_revolution = 200
degrees_per_step = 360 / steps_per_revolution
microsteps_per_step = 32
microsteps_per_revolution = steps_per_revolution*microsteps_per_step*2
degrees_per_microstep = degrees_per_step / microsteps_per_step
# Set up GPIO
PUL_min = 18
DIR_min = 23
ENA_min = 24


# PIGPIOD bootstrap
# Try to start pigpiod locally
try:
    run(['pigpiod'], timeout=3)
    pi = pigpio.pi()
except FileNotFoundError:
    # pigpiod is not installed on this system, try connecting to remote instance
    pi = pigpio.pi('192.168.137.61', 8888)

if not pi.connected:
    raise Exception("Need to have pigpiod running")

pi.set_mode(PUL_min, pigpio.OUTPUT)
```

```python
pi.write(PUL_min, pigpio.LOW)
pi.set_mode(DIR_min, pigpio.OUTPUT)
pi.set_mode(ENA_min, pigpio.OUTPUT)
pi.write(ENA_min, pigpio.HIGH)


class AntennaThread(threading.Thread):

    def __init__(self, response_queue, event_flag, duration, degrees, bearing,
    ↪  reset=None):

        # Set up thread
        super().__init__()

        module_logger.info("Starting Stepper Thread")

        self.daemon = True
        self._response_queue = response_queue
        self._event_flag = event_flag
        self._duration = duration
        self._degrees = degrees
        self._bearing = bearing
        self._reset = reset

    def run(self):
        global bearing_current

        module_logger.info("Executing Stepper Thread")

        # Point the antenna in the right direction
        AntennaThread.reset_antenna(self._bearing, self._degrees)

        # Indicate readiness
        self._response_queue.put('r')

        # Wait for the synchronization flag
        module_logger.info("Waiting for synchronization flag")
        self._event_flag.wait()

        _start_time, _stop_time = self.rotate(self._degrees, self._duration)
        bearing_current += self._degrees

        module_logger.info("Rotated antenna {} degrees for {:.2f}s"
                           .format(self._degrees, _stop_time - _start_time))

        # Put results on queue
        self._response_queue.put((_start_time, _stop_time))

        # Pause for a moment to reduce drift
        time.sleep(.5)

        if self._reset is not None:
```

```python
            # Reset antenna for next test, assuming next test has same width as current
            AntennaThread.reset_antenna(self._reset, self._degrees)

    @staticmethod
    def reset_antenna(bearing=bearing_default, degrees=0):
        global bearing_current

        _travel = AntennaThread.determine_best_path(bearing, degrees)

        # Check to see if new bearing is within 0.1
        if not math.isclose(bearing_current, bearing, abs_tol=0.1) and _travel != 0:
            _travel_duration = RESET_RATE[abs(_travel)]
            module_logger.info(
                "Resetting antenna {} degrees (from {} to {})".format(_travel, bearing_current, bearing_current + _travel))
            AntennaThread.rotate(_travel, _travel_duration)
            bearing_current += _travel
            return True

        return False

    @staticmethod
    def determine_best_path(new_bearing, degrees):
        """
        Return an optimized path to arrive at the provided bearing based on how far the travel is and the current state
        of the antenna.

        :param new_bearing: New bearing to set the antenna to
        :param degrees: How far will the antenna be traveling from this bearing
        :return: An optimized (equivalent) bearing to set the antenna
        """

        global bearing_current


        _edge_case = bool(new_bearing == bearing_current % 360)
        if _edge_case and (bearing_current >= bearing_max or bearing_current <= bearing_min):
            _travel = new_bearing - bearing_current
        else:
            # Use algorithm tested and optimized in tests/antenna_motion.py
            _travel = 180 - (540 + (bearing_current - new_bearing)) % 360
            _proposed_new_bearing = bearing_current + _travel
            if _proposed_new_bearing + degrees >= bearing_max:
                _travel = _travel - 360
            elif _proposed_new_bearing <= bearing_min:
                _travel = 360 - _travel

        return _travel
```

```python

    @staticmethod
    def rotate(degrees, duration):
        """
        Rotate by degrees and duration

        :param degrees: Number of degrees to rotate
        :type degrees: int
        :param duration: Time to take for rotation for 360 degrees
        :type duration: float
        :return: start, end
        :rtype: tuple
        """

        pi.wave_clear()

        if degrees < 0:
            pi.write(DIR_min, 1)
            degrees = - degrees
        else:
            pi.write(DIR_min, 0)

        _frequency = microsteps_per_revolution/duration

        if degrees > 6:
            _ramp1 = 1  # degrees
            _ramp1_frequency = _frequency / 4
            _ramp1_pulses = round(_ramp1 / degrees_per_microstep)

            _ramp2 = 1  # degrees
            _ramp2_frequency = _frequency / 2
            _ramp2_pulses = round(_ramp2 / degrees_per_microstep)

            _ramp3 = 1  # degrees
            _ramp3_frequency = 3 * _frequency / 4
            _ramp3_pulses = round(_ramp3 / degrees_per_microstep)

            _pulses = round((degrees - 2*(_ramp1 + _ramp2 + _ramp3)) / degrees_per_microstep)

            _ramp = [[_ramp1_frequency, _ramp1_pulses],
                     [_ramp2_frequency, _ramp2_pulses],
                     [_ramp3_frequency, _ramp3_pulses],
                     [_frequency, _pulses],
                     [_ramp3_frequency, _ramp3_pulses],
                     [_ramp2_frequency, _ramp2_pulses],
                     [_ramp1_frequency, _ramp1_pulses]]

        else:
            _pulses = round(degrees/degrees)
            _ramp = [[_frequency/3, _pulses]]

```

```
        _duration = 0
        for r in _ramp:
            assert r[0] > 0, "degrees: {}, duration: {}, ramp freq:
            ↪  {}".format(degrees, duration, r[0])
            assert r[1] > 0, "degrees: {}, duration: {}, ramp pulses:
            ↪  {}".format(degrees, duration, r[0])
            _duration += int(1000000 / r[0]) * r[1]

        _duration *= 2
        _duration /= 1000000

        _chain, _wid = AntennaThread.generate_ramp(_ramp)

        _time_start = time.time()
        pi.wave_chain(_chain)
        _time_end = _time_start + _duration

        while time.time() < _time_end:
            time.sleep(.1)

        try:
            for wid in _wid:
                if wid:
                    pi.wave_delete(wid)
        except pigpio.error as e:
            module_logger.error(e)

        return _time_start, _time_end

    @staticmethod
    def antenna_set_en(val):
        """
        Set the antenna enable pin
        :param val: Enable value to send
        :type val: bool
        """

        pi.write(ENA_min, val)

    @staticmethod
    def generate_ramp(ramp):
        """Generate ramp wave forms.
        ramp:  List of [Frequency, Steps]
        """
        pi.wave_clear()  # clear existing waves
        length = len(ramp)  # number of ramp levels
        wid = [-1] * length

        # Generate a wave per ramp level
        for i in range(length):
            frequency = ramp[i][0]
            micros = int(1000000 / frequency)
```

```
            wf1 = pigpio.pulse(1 << PUL_min, 0, micros)  # pulse on
            wf2 = pigpio.pulse(0, 1 << PUL_min, micros)  # pulse off
            wf = [wf1, wf2]
            pi.wave_add_generic(wf)
            wid[i] = pi.wave_create()

        # Generate a chain of waves
        chain = []
        for i in range(length):
            steps = ramp[i][1]
            x = steps & 255
            y = steps >> 8
            chain += [255, 0, wid[i], 255, 1, x, y]

        return chain, wid  # Return chain.


@atexit.register
def cleanup_gpio():
    """
    Cleanup - ensure GPIO is cleaned up properly
    """

    module_logger.info("Cleaning up GPIO")
    pi.wave_clear()
```

B.3.3 localizer/gps.py.

```
import csv
import logging
import os
import shutil
import threading
import time
from subprocess import Popen

import gpsd

module_logger = logging.getLogger(__name__)


# GPS Update frequency - Depends on hardware - eg BU-353-S4
↪  http://usglobalsat.com/store/gpsfacts/bu353s4_gps_facts.html
_gps_update_frequency = 1


def _initialize():
    # initialize GPS information
    if shutil.which("gpsd") is None:
        module_logger.warning("Required system tool 'gpsd' is not installed")
        return False
```

```python
    if shutil.which("gpspipe") is None:
        module_logger.warning("Required system tool 'gpspipe' is not installed. On Debian systems it is found in the package 'gpsd-clients'")
        return False

    gpsd.connect()

    try:
        gpsd.device()
        module_logger.info("GPS device connected: {}".format(gpsd.device()))
    except (KeyError, IndexError):
        module_logger.warning("GPS device failed to initialize, please make sure that gpsd can see gps data")
        return False

    return True


_initialize()


class GPSThread(threading.Thread):

    def __init__(self, response_queue, event_flag, duration, nmea_output, csv_output):
        """
        GPS Thread that, when started and when the flag is raised, records the time and GPS location
        """

        if not _initialize():
            raise RuntimeError("GPS Modules could not initialize")

        super().__init__()

        module_logger.info("Starting GPS Logging Thread")

        self.daemon = True
        self._response_queue = response_queue
        self._event_flag = event_flag
        self._duration = duration
        self._nmea_output = nmea_output
        self._csv_output = csv_output

    def run(self):
        module_logger.info("Executing gps thread")

        gps_sentences = {}

        # Wait for synchronization signal
        self._event_flag.wait()

```

```
        _start_time = time.time()
        gpspipe = Popen(['gpspipe', '-r', '-uu', '-o', self._nmea_output])

        # Capture gps data for <duration> seconds
        t = time.time() + self._duration
        while time.time() < t:
            gps_sentences[time.time()] = gpsd.get_current()
            time.sleep(_gps_update_frequency)

        module_logger.info("Terminating gpspipe")
        gpspipe.terminate()

        _end_time = time.time()
        module_logger.info("Captured gps data for {:.2f}s (expected
        ↪ {}s)".format(_end_time-_start_time, self._duration))

        # Set up average coordinate
        _avg_lat = 0
        _avg_lon = 0
        _avg_alt = 0
        _avg_lat_err = 0
        _avg_lon_err = 0
        _avg_alt_err = 0
        _lat_err_count = 0
        _lon_err_count = 0
        _alt_err_count = 0

        # Write GPS coordinates to CSV
        with open(self._csv_output, 'w', newline='') as nmea_csv:

            fieldnames = ['timestamp', 'lat', 'lon', 'alt', 'lat_err',
            ↪ 'lon_error', 'alt_error']
            nmea_csv_writer = csv.DictWriter(nmea_csv, dialect="unix",
            ↪ fieldnames=fieldnames)
            nmea_csv_writer.writeheader()

            for tstamp, msg in gps_sentences.items():

                _avg_lat += msg.lat
                _avg_lon += msg.lon
                _avg_alt += msg.alt

                # Retrieve error rates
                lat_err = None
                lon_err = None
                alt_err = None
                if 'y' in msg.error:
                    lat_err = msg.error['y']
                    _lat_err_count += 1
                if 'x' in msg.error:
                    lon_err = msg.error['x']
                    _lon_err_count += 1
```

```python
                if 'v' in msg.error:
                    alt_err = msg.error['v']
                    _alt_err_count += 1

                nmea_csv_writer.writerow({fieldnames[0]: tstamp,
                                          fieldnames[1]: msg.lat,
                                          fieldnames[2]: msg.lon,
                                          fieldnames[3]: msg.alt,
                                          fieldnames[4]: lat_err,
                                          fieldnames[5]: lon_err,
                                          fieldnames[6]: alt_err})

        # Finish calculating coordinate average
        try:
            _avg_lat /= len(gps_sentences)
            _avg_lon /= len(gps_sentences)
            _avg_alt /= len(gps_sentences)
            _avg_lat_err /= _lat_err_count
            _avg_lon_err /= _lon_err_count
            _avg_alt_err /= _alt_err_count
        except ZeroDivisionError:
            pass

        # Confirm capture file contains gps coordinates
        if os.path.isfile(self._nmea_output) and os.path.isfile(self._csv_output):
            module_logger.info("Successfully captured gps nmea data")
        else:
            module_logger.error("Could not capture gps nmea data")

        # send gps data back
        self._response_queue.put((_avg_lat, _avg_lon, _avg_alt, _avg_lat_err, _avg_lon_err, _avg_alt_err))
        self._response_queue.put((_start_time, _end_time))
```

B.3.4 localizer/interface.py.

```python
import atexit
import logging
import re
import shutil
import threading
import time
from subprocess import call, run, PIPE, CalledProcessError

from tqdm import tqdm

import localizer
from localizer.meta import OPTIMAL_BEACON_INT, STD_CHANNEL_DISTANCE, IEEE80211bg

module_logger = logging.getLogger(__name__)
```

```python

# Make sure required system tools are installed
if shutil.which("iwconfig") is None:
    module_logger.error("Required system tool 'iwconfig' is not installed")
    exit(1)
if shutil.which("ifconfig") is None:
    module_logger.error("Required system tool 'ifconfig' is not installed")
    exit(1)
if shutil.which("iwlist") is None:
    module_logger.error("Required system tool 'iwlist' is not installed")
    exit(1)


def set_interface_mode(iface, mode):
    """
    Uses ifconfig and iwconfig to put a device into specified mode (eg monitor, managed, etc).

    :param iface: Name of interface to set the mode on
    :type iface: str
    :param mode: New mode to set the interface to
    :type mode: str
    :return: Returns whether setting the interface mode was successful
    :rtype: bool
    """

    try:
        interfaces = get_interfaces()
        if iface not in interfaces:
            raise ValueError("Interface {} is not a valid interface; {}".format(iface, interfaces.keys()))

        module_logger.info("Enabling {} mode on {}".format(mode, iface))
        call(['ifconfig', iface, 'down'], stdout=localizer.DN, stderr=localizer.DN)
        call(['iwconfig', iface, 'mode', mode], stdout=localizer.DN, stderr=localizer.DN)
        call(['ifconfig', iface, 'up'], stdout=localizer.DN, stderr=localizer.DN)

        # Validate mode of interface
        interfaces = get_interfaces()
        if interfaces[iface]["mode"] == mode:
            module_logger.info("Finished enabling {} mode on {}".format(mode, iface))
            return True
        else:
            raise ValueError("Failed putting interface {} into {} mode; interface currently in {} mode"
                             .format(iface, mode, interfaces[iface]["mode"]))

    except (KeyError, ValueError, CalledProcessError) as e:
        module_logger.error(e)
```

```python
        return False


def get_interface_mode(iface):
    """
    Get the current mode of an interface

    :param iface: Interface to query for mode
    :type iface: str
    :return: Mode of interface
    :rtype: str
    """

    try:
        return get_interfaces()[iface]["mode"]
    except KeyError:
        module_logger.error("No interface '{}'".format(iface))
        return None


def get_interfaces():
    """
    Queries iwconfig and builds a dictionary of interfaces and their properties

    :return: A dictionary with keys as interface name (str) and value as dictionary of key/value pairs
    :rtype: dict
    """

    try:
        proc = run(['iwconfig'], stdout=PIPE, stderr=localizer.DN)

        # Loop through all the lines and build a dictionary of interfaces
        interfaces = {}
        current_interface = None
        for line in proc.stdout.split(b'\n'):
            line = line.decode().rstrip()
            if len(line.strip()) == 0:
                continue                                                  # Continue on blank lines
            if line[0] != ' ':                                            # Doesn't start with space
                current_line = line.split('  ')                           # Prepare the line
                current_interface = current_line[0]           # Parse the interface
                interfaces[current_interface] = {}                        # Set up new interface in dictionary
                line = '  '.join(current_line[1:])                        # Reset current line without interface
            if current_interface is not None:                             # Grab values and put them in dict
```

```python
                line = line.strip().lower()
                line_values = line.strip().split('  ')                     # Split by two spaces
                for value in line_values:                                   # Step through each value on the first line
                    value = value.strip()                                   # Clean up our value
                    if value.find(':') == -1:                               # Check for colon-separated values
                        interfaces[current_interface][value] = None         # Put in dict
                    else:                                                   # Put key/value in dict
                        value_split = value.split(':')
                        interfaces[current_interface][value_split[0].strip()] = value_split[1].strip()
            else:
                raise ValueError("Unexpected iwconfig response: {}".format(line))

        return interfaces

    except (ValueError, IndexError) as e:
        module_logger.error(e)
        return None


def get_first_interface():
    """
    Returns the name of the first interface, or None if none are present on the system.

    :return: First wlan interface
    :rtype: str
    """

    ifaces = get_interfaces()
    if ifaces is not None and len(list(ifaces)) > 0:
        return list(ifaces)[0]
    else:
        return None


def get_channel(iface):
    """
    Returns the channel the specified interface is on, or zero if it can't be determined

    :param iface: Interface to query for channel
    :type iface: str
    :return: Channel
    :rtype: int
    """
```

```python

    proc = run(['iwlist', iface, 'channel'], stdout=PIPE, stderr=PIPE)

    # Respond with actual
    lines = proc.stdout
    match = re.search('(?<=\(Channel\s)(\d{1,2})', lines.decode())
    if match is not None:
        return match.group()
    else:
        return 0


def set_channel(iface, channel):
    """
    Sets the channel of the specified interface

    :param iface: Interface to set the channel
    :type iface: str
    :param channel: Channel number to set the interface to
    :type channel: str
    :return: True for success, False for failure
    :rtype: bool
    """

    try:
        call(['iwconfig', iface, 'channel', channel], stdout=localizer.DN,
        ↪ stderr=localizer.DN)
        return True
    except CalledProcessError:
        return False


class ChannelThread(threading.Thread):
    def __init__(self, event_flag, iface, duration, hop_int=OPTIMAL_BEACON_INT,
    ↪ response_queue=None, distance=STD_CHANNEL_DISTANCE, init_chan=None,
    ↪ channels=IEEE80211bg):
        """
        Wait for commands on the queue and asynchronously change channels of
↪   wireless interface with specified timing.

        :param command_queue queue.Queue: A queue to read commands in the format
↪   (iface, iterations, hop_int)
        :param channels list[int]: A list of channels to iterate over
        """

        super().__init__()

        module_logger.info("Starting Channel Hopper Thread")

        self.daemon = True
        self._event_flag = event_flag
        self._iface = iface
```

```python
        self._duration = duration
        self._hop_int = hop_int
        self._distance = distance
        self._response_queue = response_queue
        self._channels = channels

        # Validate initial channel, if given
        self._init_chan = init_chan
        if self._init_chan and self._init_chan not in self._channels:
            raise ValueError("If you specify an initial channel, it must be in the list of channels")

        # Ensure we are in monitor mode
        if get_interface_mode(self._iface) != "monitor":
            set_interface_mode(self._iface, "monitor")
        assert(get_interface_mode(self._iface) == "monitor")

    def run(self):

        _chan_len = len(self._channels)

        # Build local channel str list for speed
        _channels = [str(channel) for channel in self._channels]

        # Initial channel position - will cycle through all in _channels
        if self._init_chan:
            _chan = self._channels.index(self._init_chan)
        else:
            _chan = 0
        set_channel(self._iface, _channels[_chan])  # Set channel to first channel

        # Wait for synchronization signal
        self._event_flag.wait()

        _start_time = time.time()
        _stop_time = _start_time + self._duration

        # Only hop channels if we have a list of channels to hop, and our duration is greater than 0
        if self._hop_int > 0 and len(self._channels) > 1:

            # HOP CHANNELS https://github.com/elBradford/snippets/blob/master/chanhop.sh
            while _stop_time > time.time():
                time.sleep(self._hop_int)
                _chan = (_chan + self._distance) % _chan_len
                set_channel(self._iface, _channels[_chan])

        else:
            time.sleep(_stop_time - time.time())

```

```
        _end_time = time.time()

        if self._response_queue is not None:
            self._response_queue.put((_start_time, _end_time))
        module_logger.info("Hopped {} channels for {:.2f}s (expected {}s)"
                           .format(len(self._channels), _end_time-_start_time, self._duration))


@atexit.register
def cleanup():
    """
    Cleanup - ensure all devices are no longer in monitor mode
    """

    ifaces = get_interfaces()
    ifaces_to_cleanup = [iface for iface in ifaces if ifaces[iface]["mode"] == "monitor"]

    if ifaces_to_cleanup:
        module_logger.info("Cleaning up all monitored interfaces")
        for iface in tqdm(ifaces_to_cleanup, desc="{:<35}".format("Restoring ifaces to managed mode")):
            set_interface_mode(iface, "managed")
```

B.3.5 localizer/process.py.

```
import csv
import logging
import os
import time
from concurrent import futures
from datetime import date

import pandas as pd
import pyshark
from dateutil import parser
from geomag import WorldMagneticModel
from tqdm import tqdm

from localizer import locate
from localizer.meta import meta_csv_fieldnames, capture_suffixes, required_suffixes

module_logger = logging.getLogger(__name__)


def process_capture(meta, path, write_to_disk=False, guess=False, clockwise=True, macs=None):
    """
    Process a captured data set
```

```
    :param meta:            meta dict containing capture results
    :param write_to_disk:   bool designating whether to write to disk
    :param guess:           bool designating whether to return a table of guessed
    ↪ bearings for detected BSSIDs
    :param clockwise:       direction antenna was moving during the capture,
    :param macs:            list of macs to filter on
    :return: (_beacon_count, _results_path):
    """

    module_logger.info("Processing capture (meta: {})".format(str(meta)))

    _beacon_count = 0
    _beacon_failures = 0

    # Correct bearing to compensate for magnetic declination
    _declination = WorldMagneticModel()\
        .calc_mag_field(float(meta[meta_csv_fieldnames[6]]),
                        float(meta[meta_csv_fieldnames[7]]),
                        date=date.fromtimestamp(float(meta["start"])))\
        .declination

    # Read results into a DataFrame
    # Build columns
    _default_columns = ['capture',
                        'pass',
                        'duration',
                        'hop-rate',
                        'timestamp',
                        'bssid',
                        'ssid',
                        'encryption',
                        'cipher',
                        'auth',
                        'ssi',
                        'channel',
                        'bearing_magnetic',
                        'bearing_true',
                        'lat',
                        'lon',
                        'alt',
                        'lat_err',
                        'lon_error',
                        'alt_error',
                        ]

    _rows = []
    _pcap = os.path.join(path, meta[meta_csv_fieldnames[16]])

    # Build filter string
    _filter = 'wlan[0] == 0x80'
    # Override any provide mac filter list if we have one in the capture metadata
    if meta_csv_fieldnames[19] in meta and meta[meta_csv_fieldnames[19]]:
```

```
        macs = [meta[meta_csv_fieldnames[19]]]
    if macs:
        _mac_string = ' and ('
        _mac_strings = ['wlan.bssid == ' + mac for mac in macs]
        _mac_string += ' or '.join(_mac_strings)
        _mac_string += ')'
        _filter += _mac_string

    packets = pyshark.FileCapture(_pcap, display_filter=_filter,
    ↪  keep_packets=False, use_json=True)

    for packet in packets:

        try:
            # Get time, bssid & db from packet
            pbssid = packet.wlan.bssid
            ptime = parser.parse(packet.sniff_timestamp).timestamp()
            pssid = next((tag.ssid for tag in packet.wlan_mgt.tagged.all.tag if
            ↪  hasattr(tag, 'ssid')), None)
            pssi = int(packet.wlan_radio.signal_dbm) if
            ↪  hasattr(packet.wlan_radio, 'signal_dbm') else
            ↪  int(packet.radiotap.dbm_antsignal)
            pchannel = next((int(tag.current_channel) for tag in
            ↪  packet.wlan_mgt.tagged.all.tag if hasattr(tag,
            ↪  'current_channel')), None)
            if not pchannel:
                pchannel = int(packet.wlan_radio.channel) if
                ↪  hasattr(packet.wlan_radio, 'channel') else
                ↪  int(packet.radiotap.channel.freq)

            # Determine AP security, if any
            ↪  https://ccie-or-null.net/2011/06/22/802-11-beacon-frames/
            pencryption = None
            pcipher = None
            pauth = None

            _cipher_tree = None
            _auth_tree = None

            # Parse Security Details
            # Check for MS WPA tag
            _ms_wpa = next((i for i, tag in
            ↪  enumerate(packet.wlan_mgt.tagged.all.tag) if hasattr(tag,
            ↪  'wfa.ie.wpa.version')), None)
            if _ms_wpa is not None:
                pencryption = "WPA"

                if hasattr(packet.wlan_mgt.tagged.all.tag[_ms_wpa].wfa.ie.wpa,
                ↪  'akms.list'):
                    _auth_tree = packet.wlan_mgt.tagged.all.tag[_ms_wpa].wfa.ie.
                    ↪  wpa.akms.list.akms_tree

```

```
            if hasattr(packet.wlan_mgt.tagged.all.tag[_ms_wpa].wfa.ie.wpa, 'ucs.list'):
                _cipher_tree = packet.wlan_mgt.tagged.all.tag[_ms_wpa].wfa.ie.wpa.ucs.list.ucs_tree

        # Check for RSN Tag
        _rsn = next((i for i, tag in enumerate(packet.wlan_mgt.tagged.all.tag) if hasattr(tag, 'rsn')), None)
        if _rsn is not None:
            pencryption = "WPA"

            if hasattr(packet.wlan_mgt.tagged.all.tag[_rsn].rsn, 'akms.list') and _auth_tree is None:
                _auth_tree = packet.wlan_mgt.tagged.all.tag[_rsn].rsn.akms.list.akms_tree

            if hasattr(packet.wlan_mgt.tagged.all.tag[_rsn].rsn, 'pcs.list') and _cipher_tree is None:
                _cipher_tree = packet.wlan_mgt.tagged.all.tag[_rsn].rsn.pcs.list.pcs_tree

        # Parse _auth_tree
        if _auth_tree:
            try:
                _type = _auth_tree.type == '2'
            except AttributeError:
                _type = next((_node.type for _node in _auth_tree if hasattr(_node, 'type') and (_node.type == '2' or _node.type == '3')), False)

            if _type == '3':
                pauth = "FT"
            elif _type == '2':
                pauth = "PSK"

        # Parse _cipher_tree
        if _cipher_tree:
            _types = []
            try:
                _types.append(_cipher_tree.type)
            except AttributeError:
                _types += [_node.type for _node in _cipher_tree if hasattr(_node, 'type')]

            if _types:
                _types_str = []
                for _type in _types:
                    if _type == '4':
                        _types_str.append("CCMP")
                    elif _type == '2':
                        _types_str.append("TKIP")
```

```
                    pcipher = "+".join(_types_str)

            if not pencryption:
                # WEP
                pencryption = "WEP" if packet.wlan_mgt.fixed.all.capabilities_tree.has_field("privacy") and packet.wlan_mgt.fixed.all.capabilities_tree.privacy == 1 else "Open"
                if pencryption == "WEP":
                    pcipher = "WEP"

        except AttributeError as e:
            module_logger.warning("Failed to parse packet: {}".format(e))
            _beacon_failures += 1
            continue

        # Antenna correlation
        # Compute the timespan for the rotation, and use the relative packet time to determine
        # where in the rotation the packet was captured
        # This is necessary to have a smooth antenna rotation with microstepping
        total_time = float(meta["end"]) - float(meta["start"])
        pdiff = ptime - float(meta["start"])
        if pdiff <= 0:
            pdiff = 0

        cw = 1 if clockwise else -1

        pprogress = pdiff / total_time
        pbearing_magnetic = (cw * pprogress * float(meta["degrees"]) + float(meta["bearing"])) % 360
        pbearing_true = (pbearing_magnetic + _declination) % 360

        _rows.append([
            meta[meta_csv_fieldnames[0]],
            meta[meta_csv_fieldnames[1]],
            meta[meta_csv_fieldnames[4]],
            meta[meta_csv_fieldnames[5]],
            ptime,
            str(pbssid),
            str(pssid),
            pencryption,
            pcipher,
            pauth,
            pssi,
            pchannel,
            pbearing_magnetic,
            pbearing_true,
            meta[meta_csv_fieldnames[6]],
            meta[meta_csv_fieldnames[7]],
            meta[meta_csv_fieldnames[8]],
            meta[meta_csv_fieldnames[9]],
```

```
            meta[meta_csv_fieldnames[10]],
            meta[meta_csv_fieldnames[11]],
        ])

        _beacon_count += 1

    _results_df = pd.DataFrame(_rows, columns=_default_columns)
    # Add mw column
    _results_df.loc[:, 'mw'] = dbm_to_mw(_results_df['ssi'])
    module_logger.info("Completed processing {} beacons ({} failures)".format(_beacon_count, _beacon_failures))

    # If asked to guess, return list of bssids and a guess as to their bearing
    if guess:
        _columns = ['ssid', 'bssid', 'channel', 'security', 'strength', 'method', 'bearing']
        _rows = []

        with futures.ProcessPoolExecutor() as executor:

            _guess_processes = {}

            for names, group in _results_df.groupby(['ssid', 'bssid']):
                _channel = group.groupby('channel').count()['capture'].idxmax()
                _encryption = pd.unique(group['encryption'])[0]
                # _cipher = pd.unique(group['cipher'])[0]
                # _auth = pd.unique(group['auth'])[0]
                _strength = group['ssi'].max()
                if not names[0]:
                    names = ('<blank>', names[1])

                _row = [names[0], names[1], _channel, _encryption, _strength]
                _guess_processes[executor.submit(locate.interpolate, group, int(meta['degrees']))] = _row

            for future in futures.as_completed(_guess_processes):
                _row = _guess_processes[future]
                _guess, _method = future.result()
                _rows.append(_row + [_method, _guess])

            guess = pd.DataFrame(_rows, columns=_columns).sort_values('strength', ascending=False)

    # If a path is given, write the results to a file
    if write_to_disk:
        _results_path = os.path.join(path, time.strftime('%Y%m%d-%H-%M-%S') + "-results" + ".csv")
        _results_df.to_csv(_results_path, sep=',', index=False)
        module_logger.info("Wrote results to {}".format(_results_path))
        write_to_disk = _results_path

    return _beacon_count, _results_df, write_to_disk, guess
```

```python


def _check_capture_dir(files):
    """
    Check whether the list of files has the required files in it to be considered a capture directory

    :param files: Files to check
    :type files: list
    :return: True if the files indicate a capture path, false otherwise
    :rtype: bool
    """

    for suffix in required_suffixes.values():
        if not any(file.endswith(suffix) for file in files):
            return False

    return True


def _check_capture_processed(files):
    """
    Check whether the list of files has already been processed

    :param files: Files to check
    :type files: list
    :return: True if the files indicate a capture has been processed already, false otherwise
    :rtype: bool
    """

    if any(file.endswith(capture_suffixes["results"]) for file in files):
        return True

    return False


def _get_capture_meta(files):
    """
    Get the capture meta file path from list of files

    :param files: Files to check
    :type files: list
    :return: Filename of meta file
    :rtype: str
    """

    for file in files:
        if file.endswith(capture_suffixes["meta"]):
            return file

    return None
```

```python


def process_directory(macs=None, clockwise=True):
    """
    Process entire directory - will search subdirectories for required files and process them if not already processed

    :param macs: list of mac addresses to filter on
    :type macs: list[str]
    :param clockwise: Direction of antenna travel
    :type clockwise: bool
    :return: The number of directories processed
    :rtype: int
    """

    # Walk through each subdirectory of working directory
    module_logger.info("Building list of directories to process")

    with futures.ProcessPoolExecutor() as executor:

        _processes = {}
        _results = 0

        for root, dirs, files in os.walk(os.getcwd()):
            if not _check_capture_dir(files):
                continue
            elif _check_capture_processed(files):
                continue
            else:
                # Add meta file to list
                _file = _get_capture_meta(files)
                assert _file is not None
                _path = os.path.join(root, _file)

                with open(_path, 'rt') as meta_csv:
                    _meta_reader = csv.DictReader(meta_csv, dialect='unix')
                    meta = next(_meta_reader)

                _processes[executor.submit(process_capture, meta, root, True, False, clockwise, macs)] = _path

        print("Found {} unprocessed data sets".format(len(_processes)))

        if _processes:
            with tqdm(total = len(_processes), desc = "Processing") as _pbar:
                for future in futures.as_completed(_processes):
                    _beacon_count, _, _, _ = future.result()
                    _results += _beacon_count
                    _pbar.update(1)

                print("Processed {} packets in {} directories".format(_results, len(_processes)))
```

```python


def dbm_to_mw(dbm):
    return 10**(dbm/10)
```

Appendix C. Utilities

C.1 Sigmoid Model: model.py

```python
from matplotlib import pyplot as plt
from scipy.optimize import curve_fit


def symmetric_sigmoid(x, a, b, c, d):
    return a + (b - a) / (1 + (x / c) ** d)

    #          0                   20
    #         90                    7
    #        180                    5
    #        360                    4

x = [0,90,180,360]
y = [20,10,8,6]

best_vals, _ = curve_fit(symmetric_sigmoid, x, y)

get_reset_rate = lambda x: symmetric_sigmoid(x, *tuple(best_vals))

RESET_RATE = [get_reset_rate(x) for x in range(1080)]

ax = plt.gca()
ax.plot(range(1080), RESET_RATE)
ax.set_xlim([0,1080])
plt.show()
print(best_vals)
```

C.2 Coprime Hop Interval Generator: generate_hop_int.py

```python
from math import gcd, ceil, floor

TU = 1024/1000000  # 1 TU = 1024 usec https://en.wikipedia.org/wiki/TU_(Time_Unit)
STD_BEACON_SCALE = 100
DEFAULT_START = STD_BEACON_SCALE/10
DEFAULT_END = STD_BEACON_SCALE*2

def coprime_rate_generator(start=DEFAULT_START, end=DEFAULT_END):
    """
    Generate a list of coprime rates that can be used as hop rates that minimize synchronization with standard beacon rate of 100TU

    :param target: The beacon rate to find alternative rates that are co-prime (no synchronization)
    :type target: int
    :param high_scaler: Multiplied by the target to set an upper limit
```

```python
    :type high_scaler: float
    :return: List of coprimes
    :rtype: list
    """

    _results = {}
    # Generate an upper limit
    for i in range(floor(start), ceil(end)):
        if gcd(i, STD_BEACON_SCALE) == 1:
            _results[i] = round(i*TU, 5)

    return sorted(_results.items())


# Script can be run standalone
if __name__ == "__main__":
    import argparse

    parser = argparse.ArgumentParser(description="Generate a list of coprimes")
    parser.add_argument("start",
                        help="The start of the list to search for coprimes",
                        nargs='?',
                        type=float,
                        default=DEFAULT_START)
    parser.add_argument("end",
                        help="The end of the list to search for coprimes",
                        nargs='?',
                        type=float,
                        default=DEFAULT_END)
    arguments = parser.parse_args()

    print(coprime_rate_generator(arguments.start, arguments.end))
```

Appendix D. `localizer` Capture Configurations

The following configurations were used for each of the treatments described in Chapter IV.

D.1 Treatment 1a: `discovery-duration-capture.conf`

```
[meta]
passes = 30
degrees = 360.0
bearing = 0.0
hop_int = 0.13312
process = False

[1]
duration = 5
capture = discovery-duration-05

[2]
duration = 10
capture = discovery-duration-10

[3]
duration = 15
capture = discovery-duration-15

[4]
duration = 30
capture = discovery-duration-30
```

D.2 Treatment 1b: `discovery-duration-capture-2.conf`

```
[meta]
passes = 45
iface =
degrees = 360.0
bearing = 0.0
hop_int = 0.18330
process = False

[1]
duration = 10
capture = discovery-duration-10

[2]
duration = 15
capture = discovery-duration-15
```

```

[3]
duration = 20
capture = discovery-duration-20

[4]
duration = 25
capture = discovery-duration-25
```

D.3 Treatment 2: `discovery-duration-fixed-capture.conf`

```
[meta]
passes = 30
degrees = 360.0
bearing = 0.0
hop_int = 0.0
process = False
channel = 8

[0]
duration = 5
capture = discovery-duration-5

[1]
duration = 6
capture = discovery-duration-6

[2]
duration = 7
capture = discovery-duration-7

[1]
duration = 8
capture = discovery-duration-8

[2]
duration = 9
capture = discovery-duration-9

[3]
duration = 10
capture = discovery-duration-10

[4]
duration = 11
capture = discovery-duration-11

[5]
duration = 12
capture = discovery-duration-12

```

```
[6]
duration = 13
capture = discovery-duration-13
```

D.4 Treatment 3: `discovery-hop-capture.conf`

```
[meta]
passes = 30
iface =
duration = 10
degrees = 360.0
bearing = 0.0
process = False

[0]
hop_int = 0.10138
capture = discovery-hop-0.10138

[1]
hop_int = 0.11162
capture = discovery-hop-0.11162

[2]
hop_int = 0.12186
capture = discovery-hop-0.12186

[3]
hop_int = 0.13210
capture = discovery-hop-0.13210

[4]
hop_int = 0.14234
capture = discovery-hop-0.14234

[5]
hop_int = 0.15258
capture = discovery-hop-0.15258

[6]
hop_int = 0.16282
capture = discovery-hop-0.16282

[7]
hop_int = 0.17306
capture = discovery-hop-0.17306

[8]
hop_int = 0.18330
capture = discovery-hop-0.18330

[9]
```

```
hop_int = 0.1935
capture = discovery-hop-0.1935

[10]
hop_int = 0.2038
capture = discovery-hop-0.2038
```

D.5 Treatment 4: `discovery-hop-dist-capture.conf`

```
[meta]
passes = 30
duration = 15
degrees = 360.0
bearing = 0.0
hop_int = 0.183296
process = False

[0]
hop_dist = 1
capture = discovery-hop-dist-1

[1]
hop_dist = 2
capture = discovery-hop-dist-2

[2]
hop_dist = 3
capture = discovery-hop-dist-3

[3]
hop_dist = 4
capture = discovery-hop-dist-4

[4]
hop_dist = 5
capture = discovery-hop-dist-5
```

D.6 Treatment 5: `capture-1-capture.conf`

```
[meta]
passes = 150
iface =
duration = 20
hop_int = 0.18330
degrees = 360.0
bearing = 0.0
process = False

[0]
```

```
test = capture-1
```

D.7 Treatment 6: `capture-2-capture.conf`

```
[meta]
passes = 150
iface =
duration = 20
hop_int = 0.18330
degrees = 360.0
bearing = 0.0
process = False

[0]
test = capture-2
```

D.8 Treatment 7: `capture-3-capture.conf`

```
[meta]
passes = 150
iface =
duration = 20
hop_int = 0.18330
degrees = 360.0
bearing = 0.0
process = False

[0]
test = capture-3
```

D.9 Treatment 8: `capture-1-focused-capture.conf`

```
[meta]
passes = 30
duration = 20
degrees = 360
bearing = 0
focused = 360,6
macs = 00:18:e7:e9:04:59,00:18:e7:e9:07:f5,00:12:17:9f:79:b6,00:16:b6:58:f3:0d,
↪  60:38:e0:06:2d:9c,60:38:e0:06:3a:d8,60:38:e0:06:34:e8,60:38:e0:06:34:ac,
↪  60:38:e0:06:3a:f0,1c:7e:e5:30:54:3e

[0]
test = capture-1
```

D.10 Treatment 9: `capture-2-focused-capture.conf`

```
[meta]
passes = 30
duration = 20
degrees = 360
bearing = 0
focused = 360,6
macs = 00:18:e7:e9:04:59,00:18:e7:e9:07:f5,00:12:17:9f:79:b6,00:16:b6:58:f3:0d,
↪ 60:38:e0:06:2d:9c,60:38:e0:06:3a:d8,60:38:e0:06:34:e8,60:38:e0:06:34:ac,
↪ 60:38:e0:06:3a:f0,1c:7e:e5:30:54:3e

[0]
test = capture-2
```

Appendix E. Additional Charts and Tables

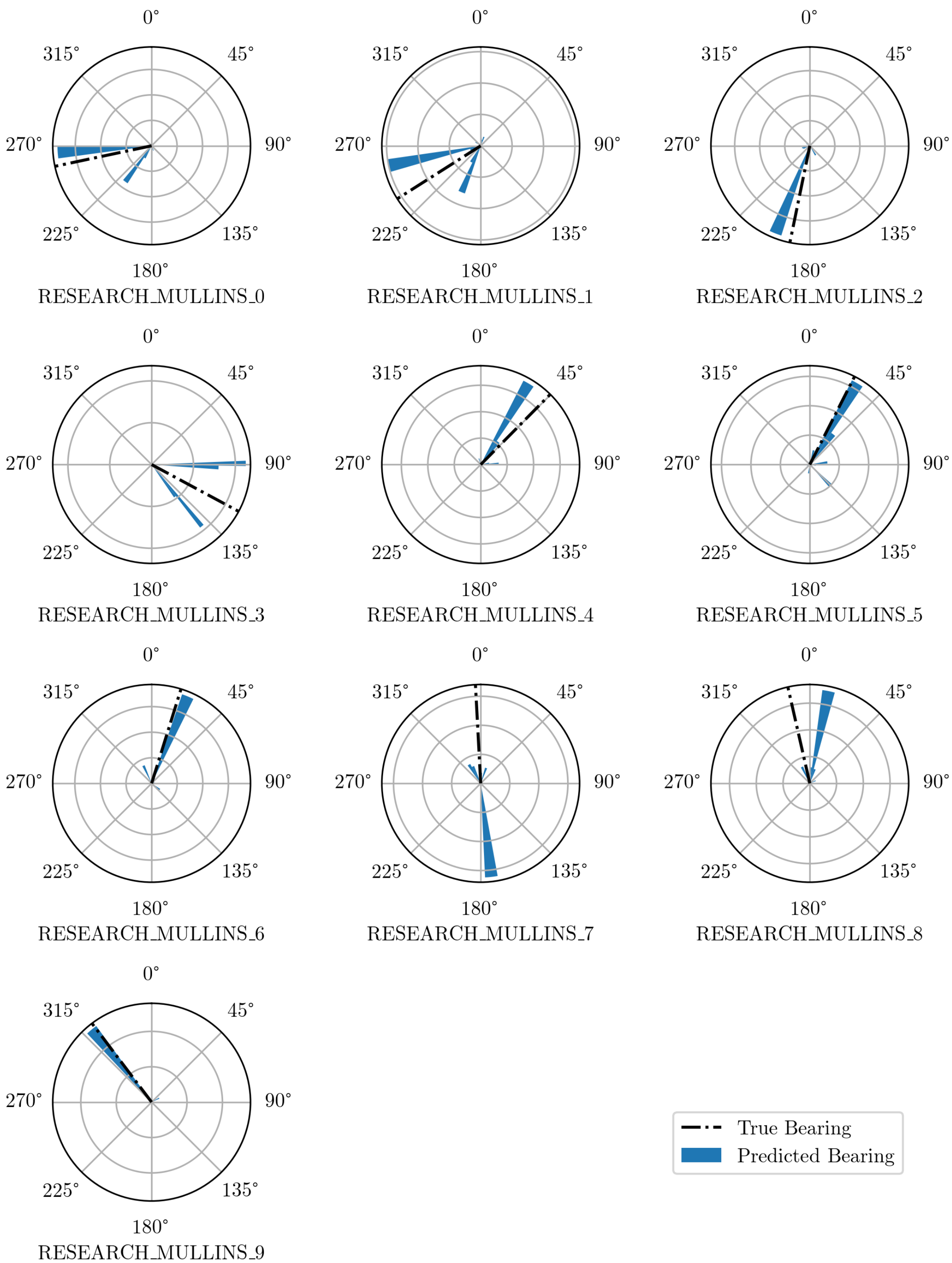


Figure 32. PCHIP Prediction Histograms Per BSSID (Treatment 5)

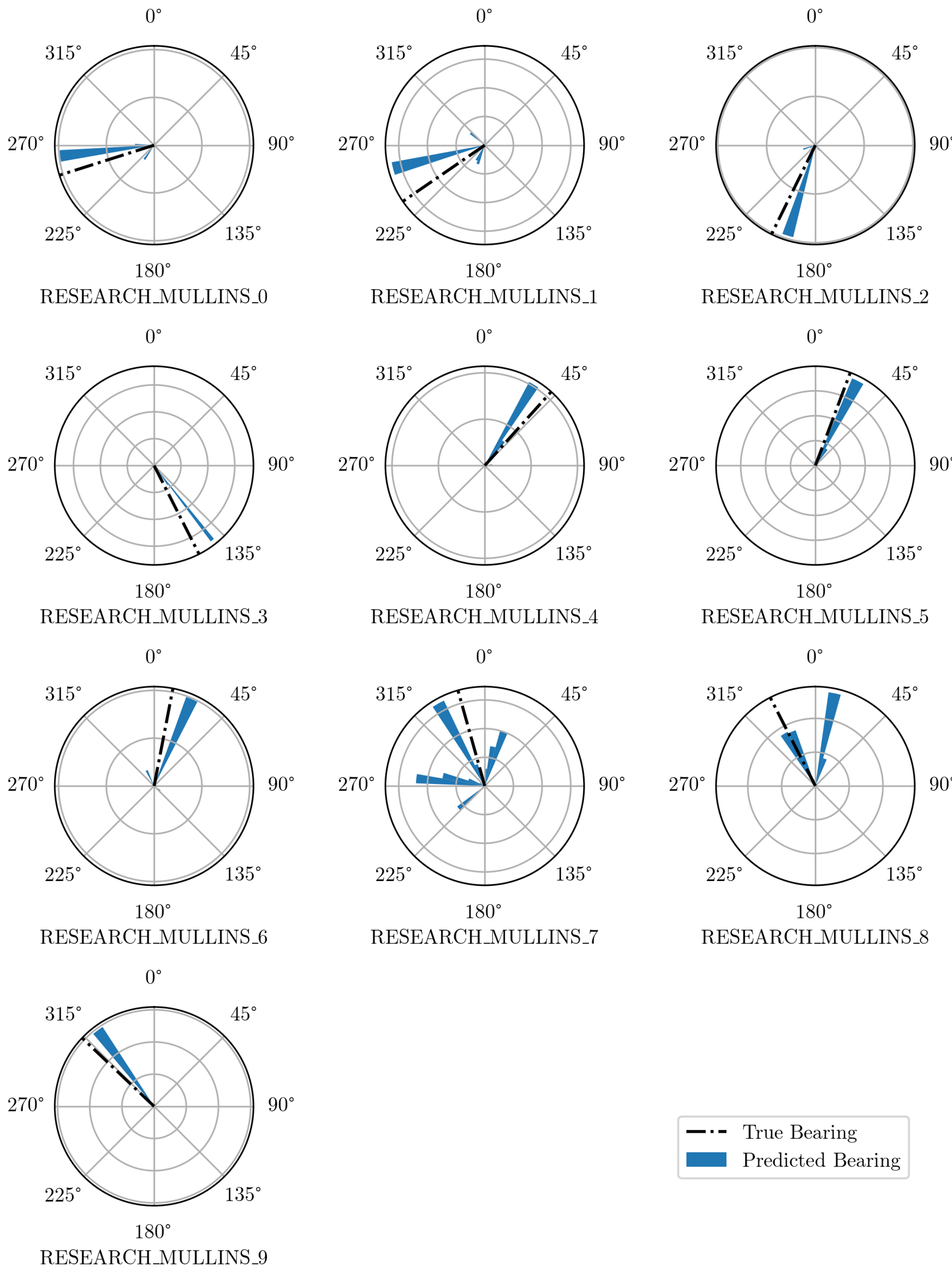


Figure 33. PCHIP Prediction Histograms Per BSSID (Treatment 6)

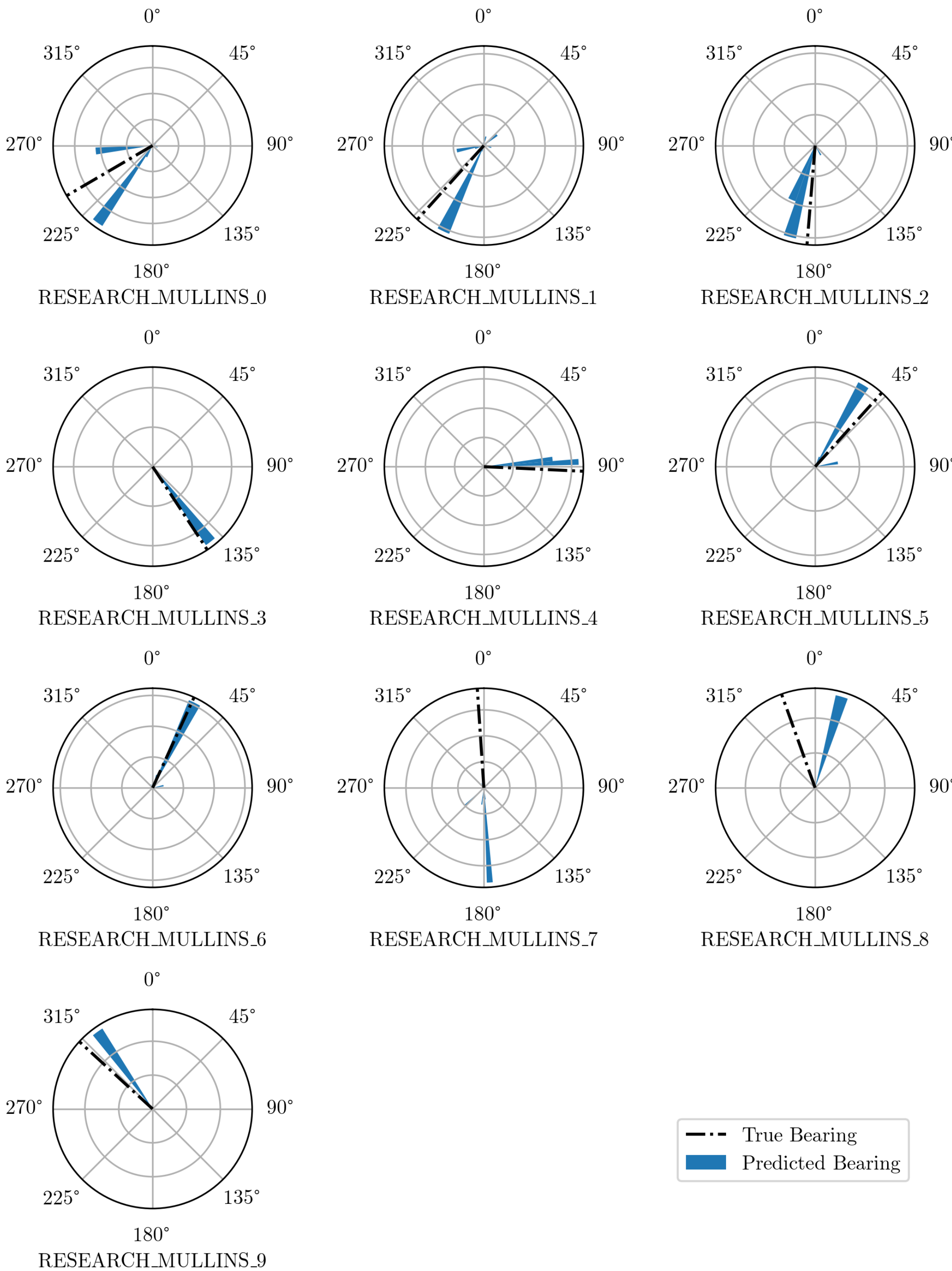


Figure 34. PCHIP Prediction Histograms Per BSSID (Treatment 7)

Table 19. Best Interpolation Method Per Sample Size

Sample Size	*Method*	*Error*
1	SLinear	26.75
2	Naive	19.44
3	PCHIP	17.44
4	PCHIP	14.08
5	PCHIP	13.08
6	PCHIP	12.09
7	PCHIP	12.60
8	PCHIP	12.60
9	PCHIP	12.70
10	BPoly	12.44
11	PCHIP	13.34
12	PCHIP	11.70
13	PCHIP	13.70
14	PCHIP	16.26
15	PCHIP	13.31
16	PCHIP	10.44
17	BPoly	8.49
18	Cubic	15.27
19	Linear	21.83
20	PCHIP	22.33
21	PCHIP	15.00

Appendix F. Least Squares Ray Optimization: `vectors.py`

This script performed least squares optimization on multiple rays to find the point closest to all rays.

```python
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from scipy.optimize import least_squares
import math

import capmap
import pandas as pd


def locate(rays):
    """
    Determine the closest point to an arbitrary number of rays, and optionally plot the results

    :param rays:    list of ray tuples (S, D) where S is the starting point & D is a unit vector
    :return:        scipy.optimize.OptimizeResult object from scipy.optimize.least_squares call
    """

    # Generate a starting position, the dimension-wise mean of each ray's starting position
    ray_start_positions = []
    for ray in rays:
        ray_start_positions.append(ray[0])
    starting_P = np.stack(ray_start_positions).mean(axis=0).ravel()

    # Start the least squares algorithm, passing the list of rays to our error function
    ans = least_squares(distance_dimensionwise, starting_P, kwargs={'rays': rays})

    return ans


def distance(P, rays):
    """
    Calculate the distance between a point and each ray

    :param P:       1xd array representing coordinates of a point
    :param rays:    list of ray tuples (S, D) where S is the starting point & D is a unit vector
    :return:        nx1 array of closest distance from point P to each ray in rays
    """
```

```
    # Generate array to hold calculated error distances
    errors = np.full([len(rays),1], np.inf)

    # For each ray, calculate the error and put in the errors array
    for i, _ in enumerate(rays):
        S, D = rays[i]
        t_P = D.dot((P - S).T)/(D.dot(D.T))
        if t_P > 0:
            errors[i] = np.linalg.norm(P - (S + t_P * D))
        else:
            errors[i] = np.linalg.norm(P - S)

    # Convert the error array to a vector (vs a nx1 matrix)
    return errors.ravel()


def distance_dimensionwise(P, rays):
    """
    Calculate the distance between a point and each ray

    :param P:       1xd array representing coordinates of a point
    :param rays:    list of ray tuples (S, D) where S is the starting point & D
    ↪   is a unit vector
    :return:        d*nx1 array of closest distance from each dimension of point
    ↪   P to each ray in rays
    """

    dims = len(rays[0][0][0])

    # Generate array to hold calculated error distances
    errors = np.full([len(rays)*dims,1], np.inf)

    # For each ray, calculate the error and put in the errors array
    for i, _ in enumerate(rays):
        S, D = rays[i]
        t_P = D.dot((P - S).T)/(D.dot(D.T))
        if t_P > 0:
            errors[i*dims:(i+1)*dims] = (P - (S + t_P * D)).T
        else:
            errors[i*dims:(i+1)*dims] = (P - S).T

    # Convert the error array to a vector (vs a nx1 matrix)
    return errors.ravel()


def plot_results(rays, ans, obj=None):
    """
    Plot the rays and the optimization results

    :param rays:    list of ray tuples (S, D) where S is the starting point & D
    ↪   is a unit vector
```

```
    :param ans:     scipy.optimize.OptimizeResult object from scipy.optimize.least_squares call
    """

    dims = len(rays[0][0][0])
    if 2 <= dims <= 3:

        # Build up a matplotlib-friendly list of coordinate arrays
        n_rays = len(rays)
        POINTS = np.empty((n_rays, dims))
        VECTORS = np.empty((n_rays, dims))

        # Get coordinates from each ray
        for i, ray in enumerate(rays):
            for dim in range(dims):
                POINTS[i, dim] = ray[0][0][dim]
                VECTORS[i, dim] = ray[1][0][dim]

        fig = plt.figure()
        gca_kwargs = {}

        quiver_args = []
        quiver_kwargs = {}

        vector_plot_args = [POINTS[:,0], POINTS[:,1]]
        vector_plot_kwargs = {'linestyle':'None', 'marker':'o', 'color':'r'}

        ans_x = ans.x.tolist()
        loc_plot_args = [ans_x[0], ans_x[1]]
        loc_plot_kwargs = {'marker':'D', 'c':'g'}

        if isinstance(obj, np.ndarray):
            object_plot_args = [obj[0][0], obj[0][1]]
            object_plot_kwargs = {'marker':'x'}

        if dims == 3:
            gca_kwargs = {'projection':'3d'}
            quiver_args = [POINTS[:,0], POINTS[:,1], POINTS[:,2], VECTORS[:,0], VECTORS[:,1], VECTORS[:,2]]
            vector_plot_kwargs['zs'] = POINTS[:,2]
            loc_plot_kwargs['zs'] = [ans_x[2]]
            if isinstance(obj, np.ndarray):
                object_plot_kwargs['zs'] = [obj[0][2]]
        else:
            quiver_args = [POINTS[:,0], POINTS[:,1], VECTORS[:,0], VECTORS[:,1]]
            quiver_kwargs['scale'] = .5

        ax = fig.gca(**gca_kwargs)
        # Plot vectors
        ax.quiver(*quiver_args, **quiver_kwargs)
        # Plot vector origins
```

```
        ax.plot(*vector_plot_args, **vector_plot_kwargs, label='Capture Location')
        # Plot calculated nearest point
        ax.scatter(*loc_plot_args, **loc_plot_kwargs, label='Prediction')

        if isinstance(obj, np.ndarray):
            # Plot object
            ax.scatter(*object_plot_args, **object_plot_kwargs, label='Emitter')

        ax.axis('scaled')
        xl = ax.get_xlim()
        yl = ax.get_ylim()
        xlen = abs(xl[0]-xl[1])
        ylen = abs(yl[0]-yl[1])

        if xlen > ylen:
            buff = (xlen - ylen)/2
            yn = (yl[0]-buff, yl[1]+buff)
            xn = xl
        else:
            buff = (ylen - xlen)/2
            xn = (xl[0]-buff, xl[1]+buff)
            yn = yl

        ax.set_xlim(xn)
        ax.set_ylim(yn)


        return ax, fig


def locate_random_rays(n=3, dims=3):
    """
    Helper function that generates random vectors to demonstrate location technique

    :param n:       The number of rays to generate
    :param dims:    The number of dimensions for each ray
    :return:        scipy.optimize.OptimizeResult object from scipy.optimize.least_squares call
    """

    from scipy.spatial.distance import cdist

    # Distance to object the rays will be point to
    dist_to_object = 50
    # Area to space the rays starting points in
    origin_area_width = 30
    # Origin point of reference
    origin = np.zeros((1,dims))

    # Generate Object Position
```

```python
    obj = origin
    while cdist(obj, origin) < dist_to_object:
        obj = np.random.randint(dist_to_object, 1.5*dist_to_object, (1,dims))

    S = []
    D = []

    # Generate S
    for i in range(n):
        s = np.full((1,dims), np.inf)
        while cdist(s, origin) > origin_area_width:
            s = np.random.randint(-origin_area_width/2,origin_area_width/2,(1,dims))
        S.append(s)

    # Generate D - Simply use the object location and add an element of random error
    for i in range(n):
        d = np.multiply(obj,np.random.uniform(.75,1.25, (1,dims)))
        d = d - origin
        D.append(d)

    rays = list(zip(S, D))

    ans = locate(rays)

    plot_results(rays, ans, obj)
    return ans


def locate_real_rays(rays, obj=None):
    ans = locate(rays)
    return plot_results(rays, ans, obj)


def bearing_to_vector(bearing):
    """
    Create a unit vector from a given bearing

    :params bearing: A float bearing
    """

    bearing = math.radians(bearing % 360)

    u = math.sin(bearing)
    v = math.cos(bearing)
    return np.array([[u,v]])

def get_point(test):
    """
    Get a test's location
    """
```

```
    # Get an object location
    _lat = pd.unique(capmap.bearings[capmap.bearings.test==test].lat_test)[0]
    _lon = pd.unique(capmap.bearings[capmap.bearings.test==test].lon_test)[0]
    return np.array([[_lat, _lon]])
```

Bibliography

[1] Tim Levin. *The Drone Noise Test*. 2017. URL: `https://www.wetalkuav.com/dji-drone-noise-test/` (visited on 01/01/2018).

[2] Ubiquiti. *UniFi AP datasheet*. 2012. URL: `https://www.ubnt.com/downloads/datasheets/unifi/UniFi_AP_DS.pdf` (visited on 01/17/2018).

[3] Ubiquiti. *airMAX Omni*. 2012. URL: `https://dl.ubnt.com/datasheets/airmaxomni/amo_ds_web.pdf` (visited on 01/17/2018).

[4] Alliance for Telecommunications Industry Solutions Inc. *Radiodetermination*. 2014. URL: `http://www.atis.org/glossary/definition.aspx?id=2554`.

[5] Ubiquiti. *airMAX Datasheet*. 2012. URL: `https://dl.ubnt.com/datasheets/airmaxyagi/airMAX_900MHz_YAGI_Antenna.pdf` (visited on 01/17/2018).

[6] Arvin Wen Tsui Tsui et al. "Accuracy performance analysis between war driving and war walking in metropolitan Wi-Fi localization". In: *IEEE Transactions on Mobile Computing* 9.11 (2010), pp. 1551–1562.

[7] Yao-Hua Ho, Yu-Ren Chen, and Ling-Jyh Chen. "Krypto: Assisting Search and Rescue Operations Using Wi-Fi Signal with UAV". In: *Proceedings of the First Workshop on Micro Aerial Vehicle Networks, Systems, and Applications for Civilian Use*. DroNet '15. New York, NY, USA: ACM, 2015, pp. 3–8. ISBN: 978-1-4503-3501-0. DOI: `10.1145/2750675.2750684`. URL: `http://doi.acm.org/10.1145/2750675.2750684`.

[8] Teemu Roos et al. "A Probabilistic Approach to WLAN User Location Estimation". In: *International Journal of Wireless Information Networks* 9.3 (2002), pp. 155–164.

[9] V Acuna et al. "Localization of WiFi Devices Using Probe Requests Captured at Unmanned Aerial Vehicles". In: *IEEE Wireless Communications and Networking Conference, WCNC* (2017). ISSN: 15253511. DOI: `10.1109/WCNC.2017.7925654`.

[10] Wei Wang et al. "Feasibility Study of Mobile Phone WiFi Detection in Aerial Search and Rescue Operations". In: *APSys* (2013). DOI: `10.1145/2500727.2500729`.

[11] Dahee Jeong, So Yeon Park, and Hyungjune Lee. "DroneNet: Network reconstruction through sparse connectivity probing using distributed UAVs". In: *IEEE International Symposium on Personal, Indoor and Mobile Radio Communications, PIMRC* 2015-Decem (2015), pp. 1797–1802. DOI: 10.1109/PIMRC.2015.7343590.

[12] Piotr Sapiezynski et al. "Opportunities and Challenges in Crowdsourced Wardriving". In: *Proceedings of the 2015 ACM Conference on Internet Measurement Conference - IMC '15* (2015), pp. 267–273. DOI: 10.1145/2815675.2815711. URL: http://dl.acm.org/citation.cfm?doid=2815675.2815711.

[13] Peter M Shipley. *Biography*. URL: http://www.dis.org/shipley/.

[14] Google. *Google Trends: Wardriving*. 2018. URL: https://trends.google.com/trends/explore?q=wardriving (visited on 01/17/2018).

[15] Kismet. *Kismet Documentation*. 2018. URL: https://raw.githubusercontent.com/kismetwireless/kismet/master/README.

[16] Kipp Jones. "What Where Wi : An analysis of information leaked by millions of wireless access points". 2015. URL: https://www.researchgate.net/publication/254080435_What_Where_Wi_An_analysis_of_information_leaked_by_millions_of_wireless_access_points.

[17] University of Washington. *Wardriving Map of Seattle 2004*. 2004. URL: http://depts.washington.edu/wifimap/maps.html (visited on 09/02/2007).

[18] Bruce Schneier. "There's No Real Difference Between Online Espionage and Online Attack". In: *The Atlantic* (Mar. 2014). URL: https://www.theatlantic.com/technology/archive/2014/03/theres-no-real-difference-between-online-espionage-and-online-attack/284233/.

[19] Hak5. *WiFi Pineapple*. 2018. URL: https://www.wifipineapple.com/ (visited on 01/01/2018).

[20] K Pelechrinis, M Iliofotou, and S V Krishnamurthy. "Denial of Service Attacks in Wireless Networks: The Case of Jammers". In: *IEEE Communications Surveys Tutorials* 13.2 (2011), pp. 245–257. ISSN: 1553-877X. DOI: 10.1109/SURV.2011.041110.00022.

[21] Pejman Najafi, Andreas Georgiou, and Dina Shachneva. "Privacy Leaks from Wi-Fi Probing". 2014. URL: http://andreasgeo.com/wp-content/uploads/2014/06/Privacy-Leaks-from-Wi-Fi-Probing.pdf.

[22] Edwin George Vattapparamban. "People Counting and occupancy Monitoring using WiFi Probe Requests and Unmanned Aerial Vehicles". In: (2016). DOI: 10.25148/etd.FIDC000246. URL: http://digitalcommons.fiu.edu/etd/2479.

[23] Cisco. *Cisco Meraki Datasheet*. 2017. URL: https://meraki.cisco.com/lib/pdf/meraki_datasheet_location_analytics.pdf (visited on 01/17/2018).

[24] Ricardo Aitken. *How much weight can delivery drones carry?* 2015. URL: https://web.archive.org/web/20170713001342/http://unmannedcargo.org/how-much-weight-can-delivery-drones-carry/ (visited on 01/01/2018).

[25] W. M. Smart. *Textbook On Sperical Astronomy*. 4th. Cambridge: Cambridge University Preses, 1949, pp. 18–19. URL: https://ia800602.us.archive.org/13/items/SphericalAstronomy/Smart-SphericalAstronomy_text.pdf.

[26] International Astronomical Union. *Astronomical Constants: Current Best Estimates*. Tech. rep. United States Naval Observatory, 2011. URL: http://maia.usno.navy.mil/NSFA/NSFA_cbe.html.

[27] David R. Williams. *Earth Fact Sheet*. 2016. URL: https://nssdc.gsfc.nasa.gov/planetary/factsheet/earthfact.html (visited on 01/01/2018).

[28] J M Chambers et al. *Graphical Methods for Data Analysis*. 1983, p. 395. ISBN: 053498052X.

[29] The Pandas Community. *pandas.Series.interpolate*. URL: https://pandas.pydata.org/pandas-docs/stable/generated/pandas.Series.interpolate.html (visited on 02/27/2018).

[30] The Scipy Community. *scipy.interpolate*. URL: https://docs.scipy.org/doc/scipy/reference/interpolate.html (visited on 02/27/2018).

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704–0188

1. REPORT DATE *(DD–MM–YYYY)*
22–03–2018

2. REPORT TYPE
Master's Thesis

3. DATES COVERED *(From — To)*
Sept 2016 — Mar 2018

4. TITLE AND SUBTITLE
Passive Radiolocation of IEEE 802.11 Emitters Using Directional Antennae

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
Law, Bradford E., Capt

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Air Force Institute of Technology
Graduate School of Engineering and Management (AFIT/EN)
2950 Hobson Way
WPAFB OH 45433-7765

8. PERFORMING ORGANIZATION REPORT NUMBER
AFIT-ENG-MS-18-M-040

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
Low-cost commodity hardware and cheaper consumer drones make the threat of home-made, inexpensive DWAPs greater than ever. Despite the vast leaps in technology these capabilities represent, UAVs are noisy and consequently difficult to conceal as they approach a potential target. This research seeks to investigate using directional antennae on DWAPs by resolving issues inhibiting directional antennae use on consumer and hobbyist drone platforms. This research presents the hypothesis that a DWAP equipped with a directional antenna can predict bearings and locations of WAPs within an acceptable margin of error.
A hardware prototype is constructed and a software framework (localizer) is built to capture data to determine optimal parameters and measure bearing and location prediction accuracy. Bearing prediction is accurate to within 14°. For location, a least-squares optimization of multiple rays is used to predict the location of WAPs and is accurate within $60\,\mathrm{m}$; an in-depth analysis of these results is presented.

15. SUBJECT TERMS
Radiolocation, UAV, Directional Antenna, Wireless Networking, Offensive Cyber Operations

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Dr.Barry Mullins, AFIT/ENG
					19b. TELEPHONE NUMBER *(include area code)*
U	U	U	UU	204	(937) 255-3636 x7979; barry.mullins@afit.edu