Aerial Software Defined Radio

MQP #: MQP-AWI-ASDR

by

Narut Akadejdechapanich
Scott Iwanicki
Max Li
Kyle Piette
Jonas Rogers

A Major Qualifying Project
Submitted to the Faculty of the

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the
Degree of Bachelor of Science
in
Electrical and Computer Engineering

December 2016
APPROVED:

Professor Alex Wyglinski

Project Advisor

Abstract

This project focuses on incorporating two new technologies, drones and software defined radios, to detect and localize relevant wireless signals with increased maneuverability. The objective lies in building a platform for wireless signal mapping for search and rescue purposes. The platform prototype will be a baseline for future development through the recommendations provided in this report.

Acknowledgements

The MASDR team have many people to thank for their unwavering assistance on helping the project become what it is now. First of all, we would like to thank Professor Wyglinski for all of his help and enthusiasm on our project. In addition to that, we would like to thank him for letting us test our drone on his property and assisting us with the paperwork in order for us to test at WPI. We would also like to thank Jon Peck of Gryphon Sensors for his support and guidance in providing us with products that would make our project a success. In addition to ordering those products, we thank Cathy Emmerton of the ECE office for her prompt ordering of anything that we needed. Lastly, we would especially love to thank Mead Landis for his kind help and guidance in making a wonderful mounting system.

Authorship

This report was a collaborative effort between all five authors. All members contributed equally to the research and development of this project.

Figure 1: Meet the team (left to right):
Scott Iwanicki, Narut Akadejdechapanich, Max Li, Jonas Rogers, and Kyle Piette

Contents

List of Figures

List of Tables

Executive Summary

Multirotor drones are cheap, small, quick, and agile, enabling innovation across many industries in the past decade. In a similar manner, Software Defined Radios are bringing cheap, reconfigurable radios to market. The combination of the highly maneuverable drone with the software defined radio allows for a unique style of triangulation. Instead of taking samples in parallel from multiple devices, samples are taken at multiple points sequentially from one device. This is only possible with a quickly moving device like a drone and targets that move slowly, such as a stationary Wi-Fi access point or lost hiker in the woods.

By researching prior art, it was determined that similar projects have been conducted on these individual technologies, but none with them combined. Examples of prior art are [**?**] and [**?**]. The former project elaborated on the use of a UAV to perform multi-agent path planning, image processing and communication to conduct search and rescue operations. The latter project implemented a localization system using a Software Defined Radio based upon an IEEE 802.11b communications platform. Consequently, the research conducted in this project will be the foundation of future development.

The implementation of this system can be separated into the following blocks: coding framework, drone control, spectrum sensing, spectrum localization, and transmission to ground. These blocks are described in Figure 2. Due to the mod-

ular platform's independence from the drone, the coding framework describes the actions of the system dependent on the state of the drone. These actions are condensed into the states of sensing, rotation, on-board computing, and transmission to ground. The states are processed by the Single Board Computer (SBC) component of the system. Most commercial drones operate at 2.4 GHz, the targeted sensing spectrum for this project. This poses an issue because of the possibility of drone control signals interfering with the signals that the SDR is receiving. To address this, the wireless cards were replaced so that these drones can operate at 5 GHz, a different frequency to the targeted spectrum. To conduct this spectrum sensing, the SBC had to be interfaced with the Universal Software Radio Peripheral hardware driver (UHD) to the Ettus board. This required knowledge of how these libraries worked and how information is parsed in order to perform matched filtering. The board also was interfaced with a Global Positioning System (GPS) that determines the device's location. This provides the necessary base coordinates for localization. To improve accuracy, the GPS was passed through a Kalman Filter to smooth the location information. GNU Radio was used to transmit this information and receive at a ground station over 900MHz.

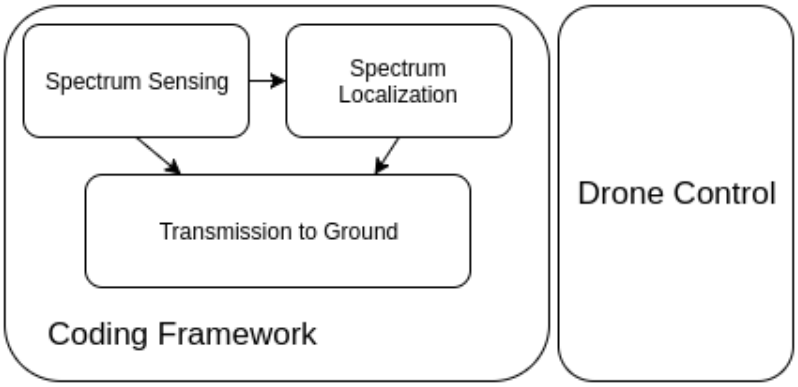


Figure 2: Block diagram of system operation. Split into drone control and SBC coding framework.

The outcome of this project was a 3DR Solo drone capable of carrying the

platform to detect nearby wireless signals. The prototype is shown in Figure 3 below. The received data corresponds well with the tests conducted. However, the

Figure 3: The complete system mounted on the drone. In flight testing was done with this setup.

obtained GPS values are not reliable enough to accurately localize the signal, despite the modeling of the Kalman Filter. Once the GPS values are reliable enough, the Python script coded will be able to map and determine the most likely point of origin of the signal. The transmission to ground is viable in lab scenarios but has not been fully integrated into the system when flown due to a malfunctioning device.

Chapter 1

Introduction

1.1 Motivation

Wireless signal localization can be used in many areas, from surveillance to search and rescue [**?**]. In the past, these efforts have been focused largely on visual inspection. Increasingly, research has turned to using wireless signal localization to provide non line-of-sight solutions [**?**]. There are two major areas of research: the use of mass communication to inform people, and the use of new technologies to physically find people. This project focuses on the latter. There are new technologies emerging that are opening up new avenues for the location of people, namely low-cost, easy to use, highly maneuverable drones and cheap, powerful software defined radios (SDRs).

Drones and SDRs have very similar advantages in their respective industries: high flexibility and maneuverability at low costs. The motivation for this project is to integrate these technologies together to further enhance the state of the art in search, especially in areas with low visibility. Enabling a search of wireless signals opens the opportunity to locate sources otherwise impossible to locate. One such

scenario would be a hiker lost in the Sierra Nevada forest with a cell phone, but no cell signal. Such a hiker would be nearly impossible to find visibly from a helicopter over the forest, but a drone with an SDR would be able to listen to beacons from their cell phone and calculate an approximation of their location as is depicted in Figure 1.1.

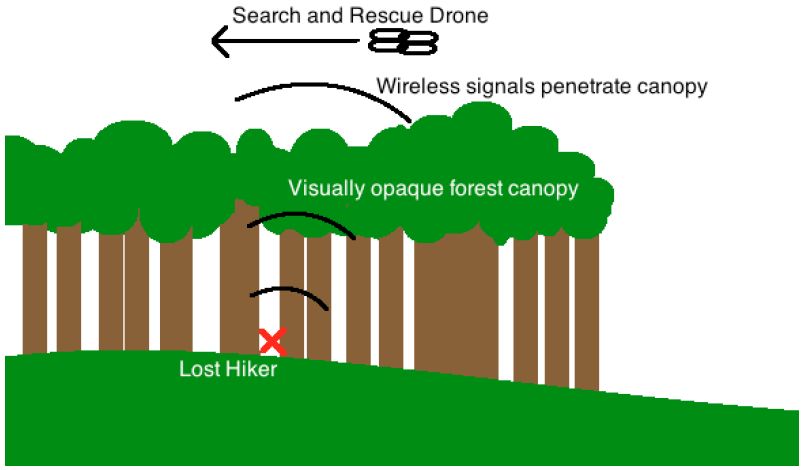


Figure 1.1: Search and Rescue Drone flying above forest canopy to locate lost hiker via cell phone beacon signals. Wireless cell phone beacons penetrate the canopy while visual inspection is blocked.

Another example scenario is the issue of locating the controller of a non-cooperative drone. Visually, this would be very difficult, but being able to sense the controlling signal and locate the controller would be possible through the use of an SDR. A diagram of this scenario is included in Figure 1.2.

1.2 Current State of the Art

The advent of small, highly maneuverable drones has allowed the expansion of search capabilities to a finer scale with the use of mounted cameras. Low cost drones are also beginning to replace helicopters for traditional search and rescue missions [?].

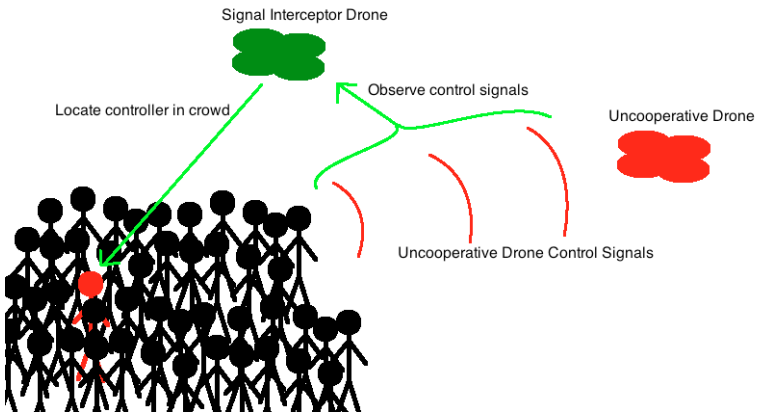


Figure 1.2: Drone Controller Localization. Someone in a crowd is controlling an uncooperative drone that shouldn't be flying. The signal interceptor drone uses the control signals to locate where in the crowd the individual controlling the drone is.

The increased maneuverability and decreased cost is driving the movement toward drones.

Another quickly growing technology is the Software Defined Radio (SDR), which is easy to reconfigure [?]. This allows an SDR to receive and transmit a wide range of waveforms. Using SDRs, its possible to create an intelligent radio that can modify its own parameters to best match its environment [?]. SDRs have also been used in a passive listening configuration to create a radar receiver [?]. The flexibility these radios provide over traditional hardware radios is perfect for any application that benefits from tuning to different frequencies, bandwidths, and modulation schemes.

Research in locating wireless emissions has produced some novel ideas using these new technologies. In a paper published in 2010, researchers note that more and more people have their cell phone on their person at all times. They investigate the detection of these phones by their wireless transmissions [?]. In another research paper, the use of SDR hardware as a radar was investigated. Through the programmability of the SDR, the researcher was able to transmit and receive radar pulses with proper time and phase coherence, properties necessary to function as

a radar receiver [**?**]. There is no mention in the published literature, however, of combining SDR technology with drone technology.

1.3 Proposed Design and Contributions

The goal of this project is to assess and demonstrate the utility of an aerially maneuverable SDR by integrating a commercially available SDR with a low cost aerial drone. By taking advantage of a drone's maneuverability, a multitude of new sampling positions are reachable, resulting in higher accuracy for locating signal sources.

The design consists of a lightweight SDR connected to a directional multi-frequency antenna. The directionality of the antenna provides the opportunity for more accurate measurements: instead of simply measuring the signal strength at multiple points to triangulate, the platform is able to identify the strength and origin direction, reducing the number of points needed for the localization of a signal source, speeding up the process. Managing the SDR and sample processing is a single board computer (SBC), chosen for its balance between low weight, low power consumption, and ample processing power. In order to allow compatability with different drone platforms, the prototype will not explicitly communicate with its carrier drone, and will therefore include its own GPS receiver and other orientation sensors. This platform will henceforth be referred to as the Modular Aerial Software Defined Radio (MASDR) platform. This project adds to the state of the art by performing research into the combination of these two newly developed technologies. The findings from this project will provide new knowledge on the ability to localize wireless signals using an aerially mounted software defined radio. Creating this MASDR platform will open the door to many more discoveries and products through further more research and development. With more time to spend on soft-

ware development, the MASDR platform would become much more capable in all aspects.

In addition to adding to the state of the art, this project seeks to develop a multipurpose platform which can be used for further research into the wireless spectrum from any location in the airspace. This could be used for channel modeling, heat mapping of wireless signals, and many more experiments into the behavior of various wireless signals in various environments.

1.4 Report Organization

This report is organized into seven chapters, the Introduction, Background, Proposed Approach, Methodology, Implementation, Results, and Conclusions. The Introduction presents the motivation for the project and the current state of the art to provide context for how this project relates to other prior and ongoing research. The Background chapter provides information on technologies and techniques used to make decisions and complete this project. In the Proposed Approach chapter, the report outlines the decision-making thought process for technical planning of the project. The Methodology chapter lays out the detailed plans of the project, its components, and the tests to verify functionality. The Methodology is followed by the Implementation chapter, which describes the implementation of the project in terms of hardware, software, and communications. In the Results chapter, the findings are presented organized by the test each finding came from. Finally, the Conclusions chapter summarizes the findings and provides recommendations for future work in the area.

Chapter 2

Digital Communications and Spectrum Sensing Overview

This chapter will introduce various fundamental topics that were taken into consideration in the design of the aerial software defined radio platform developed in this MQP. The topics proposed include spectrum allocation, energy detection, wireless transmitter localization, software defined radios, and aerial flight platforms.

2.1 Spectrum

Wireless communications are transmitted through the use of the electromagnetic spectrum. The spectrum consists of a multitude of frequency bands that have been allocated for specific uses. This section will explain how the spectrum is allocated, as well as details about the uses that are more relevant to this project.

2.1.1 Spectrum Usage

The spectrum of radio frequencies used for wireless communications is managed by the government to promote efficient use and net social benefit. In the United States, the National Telecommunications and Information Administration (NTIA) and the Federal Communications Commission (FCC) regulate the allocation of these frequencies as shown in Figure 2.1 [**?**]. As seen from Figure 2.1, the spectrum is

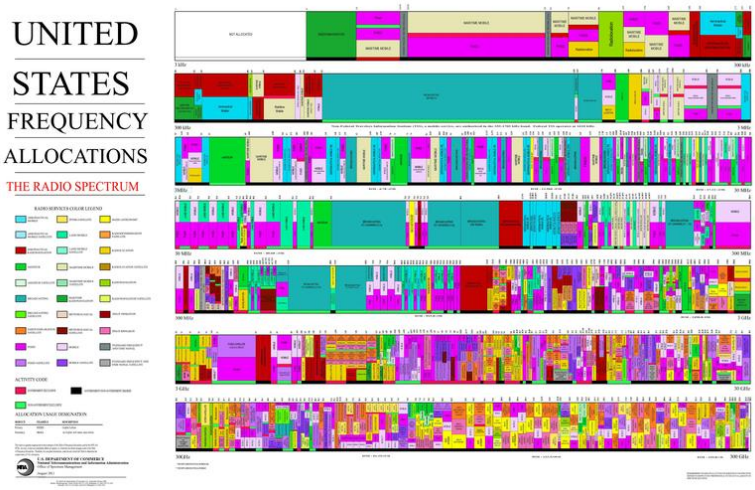


Figure 2.1: United States Frequency Allocation Chart [**?**].

divided into smaller chunks where only specified entities can utilize legally. The usage of each chunk is determined by whether the NTIA and FCC require a license to use the band or not. Only government licensed operators can communicate in the licensed bands, including AM broadcasting, FM broadcasting, and cellular communication. However, the NTIA and FCC also have allocated unlicensed bands that are open to any entity that wants to use them, provided that they still follow certain guidelines for usage. Communication technologies that use frequencies in unlicensed bands include bluetooth and WiFi. Microwaves also use frequencies in unlicensed bands some of which were allocated for general purpose use in 1947 in response to the rise in microwave cooking technology [**?**]. These frequency bands,

which includes frequencies around 2.4 GHz, were called the industrial, scientific, and medical (ISM) radio bands. The FCC chose these frequencies because microwaves are able to better transmit heat for various foods at that frequency band than at any any other. Therefore, the FCC opened up the frequency ranges as unlicensed bands so that users could operate microwaves within their household hassle-free.

2.1.2 WiFi

Building upon this unlicensed spectrum, commercial wireless data access first came about in 1985. In the early 1990s, the Institute of Electrical and Electronics Engineers (IEEE) realized that wireless communications needed to be standardized. A committee was formed that focused on providing a reliable, fast, and robust wireless solution that would scale for years to come. To do this, the IEEE made an addition to its 802 standard that is used for local area networks, and thus the 802.11 standard was created. Since then, there have been multiple iterations of the standard, with five sub-standards established for wireless communications: 802.11a, 802.11b, 802.11g, 802.11n, and 802.11ac [**?**].

The first standard, 802.11, was originally created in 1997. The data rate was capped at 2 Mbps, and transmitters broadcasted on a frequency from 2.4 to 2.483 GHz. These transmitters used time-division duplexing (TDD) which allows them to send uplink and downlink wireless traffic on the same RF channel. The transmitters also use interference mitigation techniques such as Direct Sequence Spread Spectrum (DSSS) [**?**] and Frequency Hopping Spread Spectrum (FHSS) [**?**], which are protocols that switch wireless channels when there is other wireless interference present on that channel.

All the wireless standards use a media access layer, also known as the MAC layer [**?**]. This layer is used to assist the transmission by providing frame synchro-

nization and encryption. 802.11 uses encryption via the Wired Equivalent Privacy (WEP) standard [**?**]. WEP consists of a 40 bit key that is used in order to access communication with a Wi-Fi transceiver.

WEP encryption, dynamic frequency selection, and an OFDM preamble have all been carried from one standard to the next. However, WiFi traffic can be transmitted using a variety of bandwidths dependent on their respective 802.11 standard. Wi-Fi traffic can be transmitted using a 20 MHz, 40 MHz or 80 MHz bandwidth dependent on their 802.11 standard. This is described in Table 2.1.

Table 2.1: This table outlines the bandwidth, frequency spectrum and modulation type of each corresponding 802.11 interface, from oldest protocol to newest [**?**]. As the wifi protocols become newer, the protocols utilize wider bandwidths for communication, and offer a range of operating frequencies. MIMO-OFDM was also developed for later 802.11 protocols such as 802.11n and 802.11ac in order to support multipath communication.

Interface	Bandwidth (MHz)	Frequency Spectrum (GHz)	Modulation	Max Data Rate (Mbps)
802.11a	20	5	OFDM	52
802.11b	20	2.4	DSSS	11
802.11g	20	2.4	OFDM	54
802.11n	20, 40	2.4 / 5	MIMO-OFDM	600
802.11ac	20, 40, 80	5	MIMO-OFDM	1300

The information contained in a frame can vary depending on the standard being used. However, each standard has the same preamble that signifies the start of a transmission, and what center frequency the transmission is being brodcast on. For 802.11, a wifi transmission is concentrated in one or more 20 MHz wide wifi channels. At the 2.4 GHz band, Wi-Fi is allocated throughput 11 wireless channels ranging from 1 to 11. However, only 3 of these channels (1, 6, and 11) are used, due to each channel only being 5 MHz wide. Since transmissions on the 2.4 GHz band are 20 MHz wide, the utilized channels are spaced to mitigate co-channel interference. This is shown in Figure 2.2.

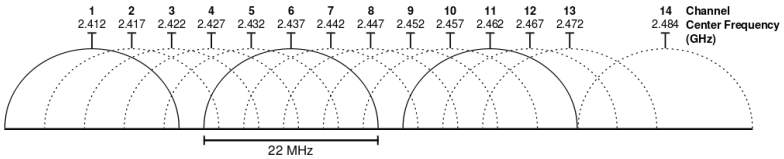


Figure 2.2: This image shows the allocation for 802.11 channels in the 2.4 GHz band [**?**]. Each channel is 20 MHz wide and also has 10 MHz allocated as a guard channel. Channels 2-5 and 7-10 are not used as a wireless channel since it would be overlapping in 2 primary wifi channels.

For the protocols 802.11a, 802.11n and 802.11 ac, the 5 GHz frequency band is also used, which contains a range of 45 channels from 36 to 165 spanning spectrally from 5180 MHz to 5825 MHz. This is shown in Figure 2.3. The detection of

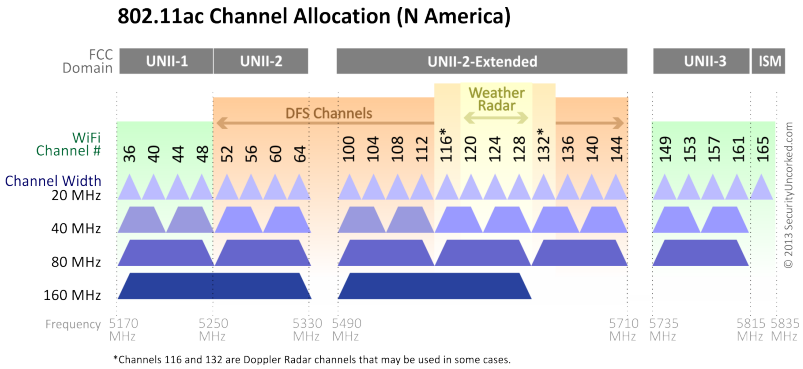


Figure 2.3: The 5GHz band allocates channels for higher bandwidth transmissions. Wi-Fi channels at 5GHz cater to transmitting information at either 20, 40 or 80 MHz. Each wifi transmission on a specified channel will adapt to a higher or lower bandwidth setting based on the corresponding signal strength and interference between a transmitter and receiver [**?**].

the OFDM PLCP preamble signifies a transmission, as well as which channel the information is being transmitted on. In addition to the preamble, you could also decode the information from the Data fields, however, all of that data is usually encrypted so that it prevents any unauthorized user from accessing it [**?**].

MAVLink

The Micro Aerial Vehicle Link (MAVLink) protocol is a free open source wireless communication protocol for controlling small unmanned aerial vehicles. This protocol first came about in 2009 by Lorenz Meier under the LGPL license [**?**]. It was designed to be a lightweight, header-only message marshaling library. MAVLink was first most commonly used with autopiloting software, but is becoming the worldwide standard in drone communication. It consists of communication between a slave drone and a master ground controller. MAVLink controllers most commonly transmit data on the 2.4 GHz band along-side 802.11 wireless communications. Therefore, an airborne receiver must be able to classify what type of signal it has received. Classifying a MAVLink transmission can be done by demodulating MAVLink's GFSK modulated data, and decoding its data frame.

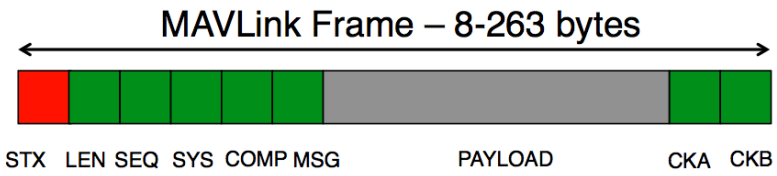


Figure 2.4: MAVLink Transmission Frame [**?**]. The red STX frame signifies the start of a transmission. This frame will always be an 8 bit code of 0xFE, and is used for start of frame detection. The following parameters give information about the payload such as it's size and it's intended recipient. The payload category contains a library-based command for the drone to perform.

As opposed to WiFi signals, MAVLink transmissions are all non-encrypted, making it more vulnerable to threats such as network attacks or an unauthorized user operating the drone. Any user can send signals to a drone using a GFSK transmitter and the drone has no knowledge on who the correct user is [**?**].

Table 2.2: This table shows the contents of a MAVLink packet. All of this information is transmitted from a base station controller to a drone that a user is operating upon. This data is not encrypted and can be sniffed by anyone who is using a MAVLink decoder.

Byte Index	Content	Value	Purpose
0	Packet start sign	0xFE	Indicates start of a new packet
1	Payload length	0 - 255	Indicates length of the following payload
2	Packet Sequence	0 - 255	Enables packet loss detection
3	System ID	1 - 225	Identifies the sending system
4	Component ID	0 - 255	Identifies the sending component
5	Message ID	0 - 255	Identifies the message being sent
6 - n+6	Data	0 - 255	Data of the message depends on message ID
n+7-n+8	Checksum		ITU hash of bytes

2.1.3 GPS

Global Positioning System, or GPS, provides highly accurate location information to a user with a receiver module. This technology was developed during the height of the Cold War in the 1960s [**?**]. GPS works by using a network of 24 satellites that are transmitting on 1575.42Mhz using a division coding scheme. The individual satellites send out information such as the current system time, and their locations. Given the locations of the satellites, as well as the calculated time of travel from the satellites to the receiver, the GPS receiver then performs localization based on this received information. The accurate time stamping required to perform this localization is only possible because each GPS satellite is capable of maintaining a highly accurate clock, and each receiver determines its current time through the interpretation of satellite data. This accurate clock information is used for the synchronization of clocks across cellular telephone systems, and other time critical applications [**?**].

2.2 Spectrum Sensing

The electromagnetic radio spectrum is becoming increasingly crowded, with this issue accelerating as the Internet of Things becomes more widespread [**?**]. With the conventional approach of spectrum management, users are assigned a specific frequency band. This method becomes less sustainable with the increasing number of users, especially in unlicensed frequency bands. In order to combat this, Cognitive Radio (or CR) was created to utilize spectrum more efficiently [**?**]. Cognitive radios are designed to provide a highly reliable connections for all users of the network, by sensing what frequencies are being used at any given moment, and utilizing the unused parts of the spectrum. Cognitive radios will be discussed in more detail in Section 2.5. The types of signals to be sensed are divided into two different groups: uncooperative users and cooperative users [**?**].

The process of spectrum sensing is made more complicated by uncertainties in the received data, including channel uncertainty and noise uncertainty. With channel uncertainty, the received signal strength can fluctuate based on characteristics of the channel, such as channel fading or shadowing. Noise uncertainty refers to the fact that the power of the noise is unknown to the receiver, making it difficult to achieve a specific sensitivity [**?**]. In order to have a functioning CR, both uncertainties need to be addressed.

As the types of signals to be sensed are split into non-cooperative and cooperative systems, so too are the methods of sensing. The present MQP involves passive sensing of the spectrum, so only the methods concerning non-cooperative systems will be discussed.

The simplest method of non-cooperative sensing is energy detection. In this approach, the power spectral density (PSD) of the received signal is taken [**?**]. The

PSD represents the measure of a signals intensity in the frequency domain computed with the fast fourier transform of a signal. This is then bandpass filtered to contain only the frequency bands being watched. These frequency bands are integrated, to determine how much energy is present in the band. If this passes a certain threshold, the frequency band is marked as occupied. A MATLAB example of energy detection is provided in Appendix **??**, and a flow diagram of energy detection is shown in Figure 2.5.

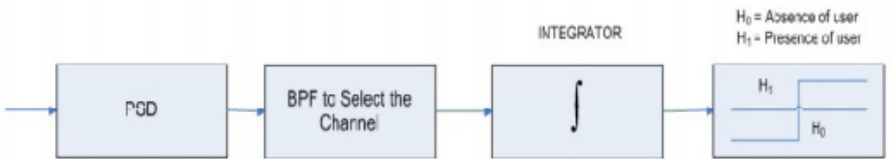


Figure 2.5: Flow diagram of energy detection. The input signal is bandpass filtered, and then summed. The signal is considered as present if the sum passes a threshold value.

This method of detection is the simplest, as it only uses the presence of the signal, ignoring key aspects of signals such as modulation method and pulse shaping. Because of this, it is the simplest detection method to implement. However, an assumption that is made while using energy detection is that the signals being searched for have significantly more power than the noise and interference of the channel. This does not always prove to be true. In addition to this, energy detection cannot be used to distinguish signals using the frequencies measured, as no pulse shape information is determined.

Another approach to spectrum sensing is through the use of matched filters. Matched filters are designed to maximize the signal to noise ratio (SNR) given an input signal, which will be called s_{in} [**?**]. To sense signals through matched filtering, prior knowledge of the reference signal (s_{ref}) to be detected must be known. s_{in} will then be correlated with s_{ref}. This produces a value m, that will be compared to

a threshold value. This process is described in Equation (2.1) below and in Figure 2.6. N represents the length of the reference signal, in samples.

$$m = \sum_{k=0}^{N} s_{in}[t-k]s_{ref}[N-k] \tag{2.1}$$

Figure 2.6: Flow diagram of matched filter. The input signal is bandpass filtered. The resulting signal is then correlated with the expected signal. If the output of this correlation passes a threshold, the signal is considered as present.

Matched filters rely on prior knowledge of the characteristics of signals to be detected, otherwise this method will not be accurate. The prior knowledge constraint limits the use of signal detection when unexpected signals are involved. Furthermore, matched filter detection is optimal with stationary Gaussian noise to hinder distortion. This will limit the real world applications for signal detection with matched filtering as most channels are time-varying and non-gaussian [**?**]. A MATLAB example of a matched filter is provided in Appendix **??**.

Cyclostationary Feature Detection (or CFD) is a method of spectrum sensing that depends on the fact that all communication schemes have some sort of signal repetition as a core aspect. A signal with this kind of repetition is called a Cyclostationary Process [**?**]. As a result of this property, when you correlate the signal with itself (or autocorrelation), there will be repeated peaks. The periodic nature of the signal also means that the autocorrelation will be periodic as well. This allows it to be expressed as a Fourier Series, called the Cyclic Autocorrelation Function (CAF) and denoted by $R_x^{\alpha}(\tau)$ [**?**].

$$R_x^{\alpha(\tau)} = lim(t- > \infty)\frac{1}{T}\int_{-T/2}^{T/2} R_x(t,\tau)exp(-j2\pi\alpha(t)dt) \tag{2.2}$$

$R_x(t,\tau)$ is the autocorrelation function of x at t with x at τ. This function gives us an understanding of the times when the signal repeats, but is missing vital information about the repeated frequencies. By taking the fourier transform of $R_x^{\alpha}(\tau)$, one can get a better understanding of the frequencies in the Cyclostationary Process. This is denoted as $S_x^{\alpha}(f)$, and is equal to Equation (2.3):

$$S_x^{\alpha}(f) = \int_{-\infty}^{\infty} R_x^{\alpha}(\tau)exp(-j2\pi f\tau d\tau) \tag{2.3}$$

This is called the Spectral Correlation Function (SCF). When it is normalized, it becomes the Spectral Coherence Function (SOF):

$$C_x^{\alpha}(f) = \frac{S_x^{\alpha}(f)}{(S_x^{\alpha}(f+\alpha/2)S_x^{\alpha}(f-\alpha/2))^{1/2}} \tag{2.4}$$

The values of the SOF ranges between 0 and 1, and represents the strength of the periodicity at that point. By plotting this value, the unique response of the Cyclostationary Process can be found, allowing for it to be categorized.

One of the major benefits of cyclostationary feature detection is that it is not nearly as affected by noise. Under the generally held assumption that the noise is white and Gaussian, there is no periodic response, and as such noise is not factored into CFD. Example plots from an implementation of CFD are shown in Figure 2.7, Figure 2.8, and Figure 2.9.

The primary downside to Cyclostationary Feature Detection is the complexity

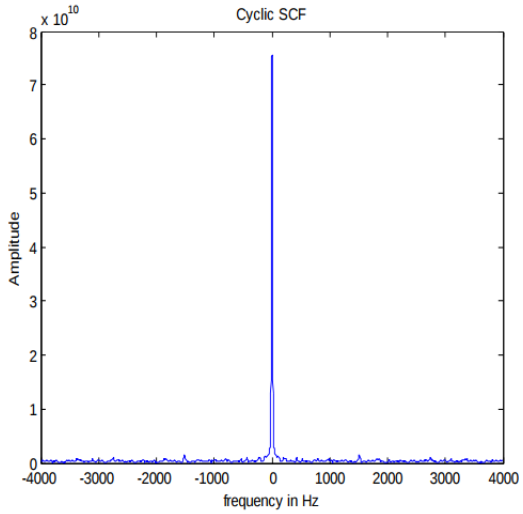


Figure 2.7: The SCF of a simple unmodulated signal. There is one peak around 0 Hz, with noise everywhere else.

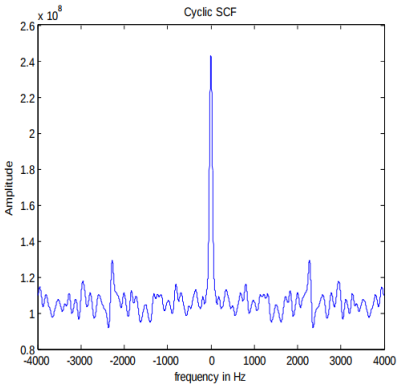


Figure 2.8: The SCF of a simple signal, modulated with BPSK. There is a unique response in addition to the peak at 0 Hz.

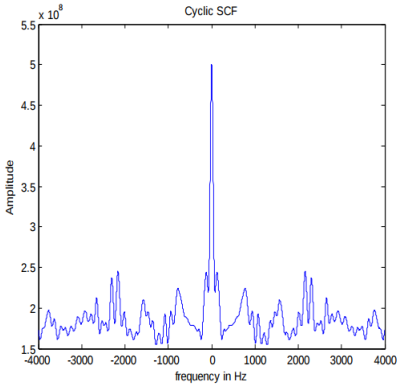


Figure 2.9: The SCF of a simple signal, modulated with QPSK. There is a response that is unique from the BPSK result.

and time required to properly utilize it. The complexity lies in the number of integrals and correlations that need to be computed in order to implement cyclostationary feature detection. In addition, the system needs to listen for signal long enough for the Cyclostationary Process to repeat. As a result of these downsides, it is not common to be used on embedded systems [**?**].

2.3 Radio Localization

Once a signal has been identified on a measured frequency, the next step is localization. Localization refers to the act of determining the location of a transmitted signal [**?**]. There are many different methods to do this. However, many of them become impractical for mobile passive sensing. In this section, a variety of localization techniques will be discussed, as well as the practicality of implementing each of them.

For each of the techniques discussed below, the same basic concept is used. Three different receivers are placed around the transmitter being localized, as shown in Figure 2.10. They will be called stationary nodes and the unknown node, respectively. Each stationary node gets information about its location relative to the unknown node simultaneously. For Time Difference of Arrival (TDoA), Time of Arrival (ToA), and Received Signal Strength (RSS) Localization, each stationary node calculates the distance to the unknown node [**?**]. This information is used with the known locations of the stationary nodes to localize the unknown node. Angle of Arrival (AoA) [**?**] functions differently and will be described in a separate section.

Table 2.3 shows the most important details of the localization techniques that will be described.

Table 2.3: A table describing localization techniques. Compares how a location is determined and if the unknown node needs to cooperate.

Localization Method	Mathematical Technique	Needs Unknown Node Participation?
TDoA	Distance	Yes
ToA	Distance	Yes
RSS	Distance	No
AoA	Angle	No

In the distance-based localization techniques, each stationary node knows the unknown node is a certain distance away [**?**], allowing for it to be anywhere along a

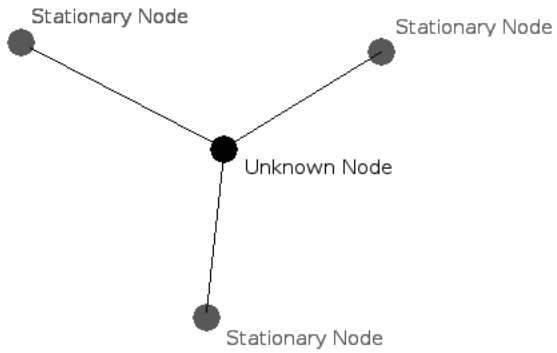


Figure 2.10: Node Localization based on three stationary nodes to detect the location of the unknown node.

circle, with the stationary node in the center. By finding out where all three circles intersect, the unknown node can be located. The distance of the unknown node from any of the stationary nodes is calculated using the following equation:

$$D_i = \sqrt{(X - x_i)^2 + (Y - y_i)^2} \tag{2.5}$$

By plugging in the locations and calculating the distance for each stationary node, a system of equations is created, making it possible to solve for the location of the unknown node.

One method of localization is Timed Difference of Arrival (TDoA) [**?**]. In this method, the unknown node transmits a signal over radio frequencies. Using the difference in the timestamp and the actual time of receiving the signal, the stationary nodes can calculate the distance between each stationary node and the unknown

node, using Equation (2.6), where c represents the speed of light.

$$d = c * (t_{actual} - t_{expected}) \tag{2.6}$$

This method of localization is dependent on three things: the synchronization of the nodes involved, the participation of the unknown node, and the existence of three or more stationary nodes. In the use case presented by this MQP, the most difficult part about this implementation is the participation of the unknown node. Since the aerial platform has no communication with the object being localized, this method is impractical.

Using the Time of Arrival (or ToA) approach is similar to TDoA in concept [**?**]. Each of the stationary nodes sends a signal at a specific time, known to the unknown node, t_0. Using the difference in t_0 and the time that the base receives the signal, or t_1, the unknown node calculates the time the signal was in the air, and uses the speed of an electromagnetic wave to determine the distance to the stationary node, as described above. With three of these calculations, the location of the unknown node can be triangulated.

Like TDoA, ToA requires the synchronization of the nodes involved [**?**]. In most implementations, this is done using digital timestamps. It also requires the participation of the unknown node. Similar to TDoA, these requirements make ToA an impractical fit for this MQP.

Unlike the previous two methods, which depend on knowing the time a signal was sent, received signal strength (RSS) localization uses the strength of a received signal to localize the unknown node [**?**]. Each of the stationary nodes receives the signal being output by the unknown node. Using Equation (2.7) and Equation (2.8), the stationary nodes can calculate the distance to the unknown node [**?**]. From this,

they can triangulate the location as described before.

$$RSSI = 10\alpha log(d) \tag{2.7}$$

$$d = 10^{(RSSI - RSSI_{calibration})(-10\alpha)} + d_{calibration} \tag{2.8}$$

Since RSS localization depends on the measurement of the signal strength, it is adversely affected by sources of noise and interference, such as channel fading and multipath interference [**?**], resulting in lower accuracy. The effects of noise and shifting channels can be somewhat mitigated by averaging the RSS data. Unlike the previously described methods, some localization techniques use the angle that the signal arrives at relative to a reference direction, or the Angle of Arrival (AoA) [**?**]. To use this method, antenna arrays or directional antennas are used to determine the angle at which the signal arrived at. Similar to TDoA, the stationary nodes wait for a signal from the unknown node. Using the directional antenna or the antenna array, the AoA is calculated at each stationary node. From each of these nodes, a ray can be drawn originating from the node that follows the angle of arrival. The unknown node is located based on where the rays intersect.

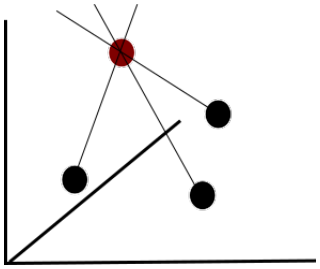


Figure 2.11: Angle of arrival diagram. A ray is drawn from each stationary node towards the unknown node, based on the signal sent out by the unknown node. The intersection of these rays is where the unknown node is located

AoA localization consists of similar issues as other techniques. Noise and channel issues still come into effect. Additional issues occur if these unknowns cannot be

dealt with. Another constraint is the requirement for highly directional sensing, which complicates the sensing of a signal before localizing it. This can either be done with many antennas or a very directional antenna, but it adds complexity.

2.4 Kalman Filtering

In the previous section, the localization techniques discussed focused on the merits of the technique itself, assuming that the numbers used in the techniques are as good as they can be. Usually, this isn't the case [**?**]. Changing environment conditions and noisy sensors introduce errors that are hard to predict, and thus hard to mitigate. This is one of the motivations of Kalman filters, as well as the field of error estimation in general [**?**]. In addition, Kalman Filters are frequently used as a way to control a system, using the estimate as feedback. Any applications relevant to this project are solely estimation-based, so the rest of this section will focus on this aspect of Kalman filters.

In most situations, the known information of a system is a combination of sensor values and a model of what the sensor values should be. The sensor values are imperfect, and have some amount of noise [**?**]. The model, based on some fundamental assumptions, is too imprecise to perfectly apply to the situation. By choosing some middle value in between the measurement and the prediction, a more accurate value can be produced. The Kalman filter is one way to decide which middle value to use [**?**]. It applies only to linear systems. Non-linear systems can be linearized and used as an input to a Kalman filter, in which case it is called an Extended Kalman filter. The math for Extended Kalman filters is fairly complicated, so the focus of this section will be standard Kalman filters.

The equations used in a Kalman filter are shown below:

Predict:

$$\vec{x} = \mathbf{F}\vec{x} + \mathbf{H}\vec{u} \tag{2.9}$$

$$\mathbf{P} = \mathbf{F}\mathbf{P}\mathbf{F}^{\mathbf{T}} + \mathbf{Q} \tag{2.10}$$

Update:

$$\vec{y} = \vec{M} - \mathbf{H}\vec{x} \tag{2.11}$$

$$\mathbf{S} = \mathbf{H}\mathbf{P}\mathbf{H}^{\mathbf{T}} + \mathbf{R}\mathbf{S} \tag{2.12}$$

$$\mathbf{K} = \mathbf{P}\mathbf{H}^{\mathbf{T}}\mathbf{S}^{-\mathbf{1}}\mathbf{K} \tag{2.13}$$

$$\vec{x} = \vec{x} + \mathbf{K}\vec{y} \tag{2.14}$$

$$\mathbf{P} = (\mathbf{I} - \mathbf{K}\mathbf{H})\mathbf{P} \tag{2.15}$$

The first set of equations predicts the current state given the last state. Equation (5.1) updates the state based on the current state and current input. The vector $\vec{x}$ is the state of the Kalman filter. This has the values being filtered. $\vec{u}$ is the value of the controls being used to alter the state. In our application, this is 0. $\mathbf{F}$ is the state transition matrix, which determines how the different states interact with each other. $\mathbf{H}$ is the control matrix. Equation (2.10) updates the covariance matrix of the state, which is $\mathbf{P}$. $\mathbf{Q}$ is the covariance of the noise of the environment.

The second set of equations updates the Kalman filter to take into account the most recent measurement. $\vec{y}$, sometimes called the innovation, is the difference between the measurement received and the last state prediction. $\mathbf{S}$ is an intermediate matrix, used so that the calculation of the Kalman gain $\mathbf{K}$ is not too complicated in one step. The calculated Kalman gains are then used to update the state and state

covariance to take into account the updated model and input.

2.5 Software Defined Radio

Due to the rapidly increasing number of users dependent on efficient spectrum allocation for communication, the use of reprogrammable radios has become an integral part of designing communication systems. Radios exist in a wide range of devices, such as cell phones, cars, televisions, garage door openers, and computers. A radio is any device that transmits or receives wireless signals that are within the radio frequency band of the electromagnetic spectrum. A software defined radio (SDR) is defined as a radio in which some or all of the physical layer functions are software defined [**?**]. Traditional hardware based radios are nearly impossible to modify post-production, and have limited ways to be repurposed. SDRs, on the other hand, are comparatively inexpensive, highly reusable, and are easily configurable to support multiple waveform standards.

There are multiple different families of SDRs, two of which are RTL and Universal Software Radio Peripheral (USRP). An RTL is a low cost SDR that uses a DVB-T TV tuner dongle based on the RTL2832U chipset [**?**]. The DVB-T TV tuner was converted to be used as a wideband SDR using a new software driver. The RTL-SDR can be used for many applications including listening to aircraft traffic control, decoding ham radio packets, and sniffing GSM signals. The USRP is a flexible transceiver developed by Ettus Research that is able to use a standard PC to prototype wireless systems. USRPs are able to prototype a wide range of single-channel and multi-input multi output (MIMO) wireless communication systems [**?**]. USRPs can be programmed using software frameworks including GNU Radio, MATLAB, Simulink, LabVIEW, and C++.

SDRs can be implemented on field programmable gate arrays (FPGA), digital signal processors (DSP), programmed System on Chips (SoC), general purpose processors (GPP), personal computers (PC), or other reprogrammable application specific integrated circuits (ASICs). The use of these reprogrammable technologies allows for the possibility of constant updates or dynamic radio systems without the need to add additional hardware. The primary goal of designing an SDR is to implement as much of the radio in the digital space, minimizing the use of analog components. The digital portion of an SDR performs the data compression, encoding, modulation, demodulation, decoding, and decompression in software. The only analog components are a digital to analog converter (DAC), an analog to digital converter (ADC), and a RF circuit. The RF circuitry on the transmitter consists of a smoothing filter to reduce the hard edges of the baseband signal that the DAC outputs as well as circuitry to upconvert the baseband signal. The RF circuitry on the receiver side consists of a downconverter to move the signal to the baseband as well as filters to remove noise from the signal. The DAC and ADC serve as the bridge between the analog and digital realms enabling the software defined signal to be converted to an analog one to transmit and receive, then converted back to digital on the receiving end shown below in Figure 2.12.

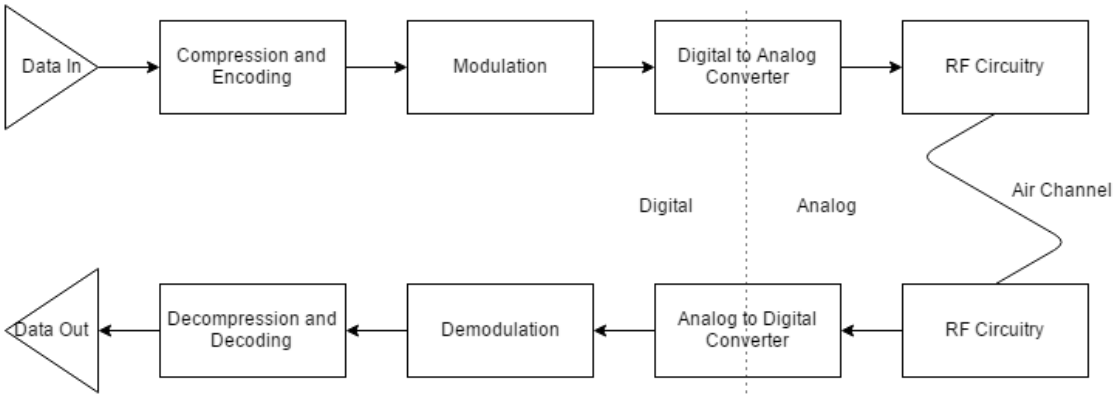


Figure 2.12: Flow diagram of an SDR with the distinction between the digital and analog components.

A typical transmitter passes a modulated signal to the DAC which takes the digital input and outputs a baseband analog signal. The analog signal is upconverted to the carrier frequency in a single step or multiple steps. The receiver then takes the received analog signal and downconverts it. It then passes the baseband analog signal to the ADC to convert the signal back into digital for the SDR to process. The capabilities of the ADC and DAC, such as the bandwidth and noise tolerance of these two components, will dictate the complexity of the RF circuitry that is needed to satisfy these requirements.

A software defined radio, such as an adaptive radio [**?**], has the capability to be much more sophisticated than its analog counterpart. Adaptive radios are able to monitor their own performance and adjust their parameters to ensure the highest quality of service [**?**]. A more advanced type of adaptive radio is the cognitive radio [**?**], mentioned in Section 2.2, which is able to monitor, sense, and detect conditions of their operating environment and adjust its characteristics to match those conditions. This allows the radio to provide improved performance and quality of service. These radios are able to find and transmit on open gaps of radio spectrum. This allows for minimal interference from other sources. Cognitive radios use the trends of the channel to determine whether to switch to low occupied channels or continue using the current channel. The most advanced type of adaptive radio is the intelligent radio. These radios are capable of machine learning, enabling it to improve its algorithm for adjusting the radio's parameters based on previous experience when changes to performance or the environment occur [**?**]. The relationship between these types of radios is shown below in Figure 2.13.

There are three major groups of algorithms that intelligent radios are based on: machine learning, genetic algorithms, and fuzzy control [**?**]. Neural networks are a type of machine learning that use large group of nodes to solve problems

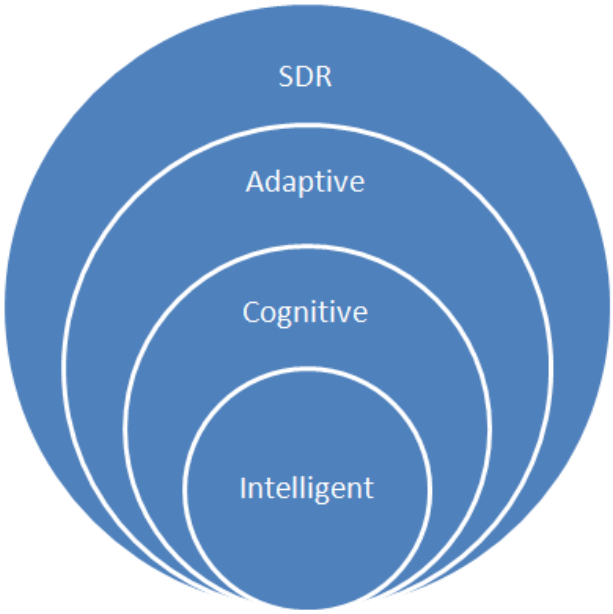


Figure 2.13: The relationship between SDR, adaptive radio, cognitive radio, and intelligent radio [?].

similarly to how the human brain operates. When new information is acquired, it is incorporated into the algorithm to improve the output [?]. This technique is used to estimate the performances of WiFi networks and respond to changes in the network. Fuzzy logic is usually combined with neural networks that adapt to the environment during the operation of an intelligent radio. A fuzzy logic control system is used to find the solution to a problem given imperfect information [?]. There are three main components in a fuzzy logic controller: fuzzifier, fuzzy logic processor, and defuzzifier. The fuzzifier maps the inputs into fuzzy sets, the fuzzy logic processor determines a fuzzy output based on a set of rules, and the defuzzifier transforms the fuzzy output into a crisp output. One of the main advantages to fuzzy logic is its low complexity, which is extremely useful in real-time radio applications. Another example of an algorithm used in intelligent SDRs is the wireless system genetic algorithm (WSGA). This algorithm is a multi-objective genetic algorithm (MOGA)

designed for the control of a radio by modeling the physical radio system as a biological organism and optimizing its performance through genetic and evolutionary processes [**?**]. Each aspect of the radio, including payload size, coding, encryption, and spreading, can be considered genes that can be modified through these processes. These genes are combined into a chromosome that is used to evolve the system. The algorithm analyzes the effectiveness of the chromosome through weighted fitness functions defined by performance evaluations of the current radio channel [**?**].

SDRs are used in a variety of different applications and are becoming more sophisticated. They are more modular than the analog equivalent and are capable of adapting to their environment. They are also very easily updated and modified as opposed to using an analog circuit. The SDR is widely used because of these benefits over the analog design.

2.5.1 Interfacing with SDRs

One development framework used was a free software called GNU Radio. GNU Radio was first released in 2001 by Eric Blossom, and is is a software defined radio framework where users continue to aid in its development, in addition to using it [**?**]. GNU Radio users throughout the years have created a number of different module blocks that are used in signal processing and programming the software defined radio. All of GNU Radios tools are used to make a GNU Radio application called a flowgraph. GNU Radios software base is written in C++, however, users have the option to create their own modules called out-of-tree modules in Python. GNU Radio makes use of a graphical interface which they call GNU Radio Companion. In GNU Radio companion you can manually connect each type of block and add or delete modules use a drag and drop type of feature. In GNU Radio each input or output has its own respective data type which are denoted by colors at their

respective input and output ports.

Although GNU Radio is appealing to users for it's power and ease of use, it can use up a substantial amount of processing power and memory due to all of the overhead the entire software has. Therefore, in order to reduce this amount of overhead, people use a more basic development environment called UHD.

UHD (or USRP Hardware Driver) is a set of C++ libraries created by Ettus Research to allow for control of their USRP SDR platform within C++ programs [**?**]. It provides a thorough set of commands with which different aspects of the SDR can be used, including synchronization, transmission, and receiving. GNU Radio, with its pre-built blocks, is built on top of UHD, using the UHD commands for lower level operation. By using the lower-level environment, there is less latency. Unlike GNU Radio, there are not very many pre-built blocks, instead focusing on having the programmer implement these details. In some cases, this is good. With many of the GNU Radio blocks, there are many sub-blocks built in, making it more difficult to understand its performance. With UHD, it is very clear what each block does, as it has to be programmed. However, this makes the development time of any system much higher. This is the main trade off between the two platforms.

2.6 Aerial Platforms

The inspiration behind this MQP was to discover how much information could be gathered from the wireless spectrum when observed from the air. Aerial platforms have been used in the past for surveying tasks such as geographical mapping [**?**]. These systems generally use a camera to capture visual images from above. To facilitate flying our sensory system, three main mobile methods of elevating the sensory unit were considered: kites, fixed-wing aircraft, and multicopters, each of

which will be described in detail below.

2.6.1 Kites

A kite is a passive structure tethered to the ground that stays airborne by catching the wind [**?**]. There are many different styles of kite, including parafoil, rokkaku, delta, and sled, but they all rely on the same principle to fly [**?**] [**?**]. Kites have nearly unlimited flight time and high payload capacity when operated in favorable conditions, but in exchange they sacrifice control, reliability, and maneuverability.

Since kites are powered by the natural phenomenon of the wind in the atmosphere, they do not require any power to stay aloft. However, depending on the style of kite, they do require specific conditions to fly. Almost all kites need at least 2-3 mph of wind to get up into the air, and delta and rokkaku are limited to a maximum of 12-16 mph [**?**]. Parafoils can fly in wind up to 20 mph, but can be unstable.

The only control a user has over a kite in flight is changing its height by letting out or reeling in more of the tether cord. Different styles of kite have different flying angles, which impacts the amount of weight they can carry. Some kites, such as the rokkaku and delta (example in Figure 2.14) styles, can be modified during assembly to change their angle of flight, where higher angles produce more lift. While parafoils can inherently carry more weight, they also have a low flying angle, which can be problematic in some scenarios. The preparation for flight involves assembling the kite, normally by inserting cross-bars, and unreeling the tether in an open space that allows the kite to take off.

Kites are, by necessity, constructed of very light materials and cannot support much extra weight in light wind situations. Again, the style of kite affects the carrying capacity: deltas can carry a moderate amount of weight, but need higher winds to stay aloft. Parafoil and rokkaku kites can carry more weight, with 10 lbs

Figure 2.14: A delta style kite. Delta style kites can be configured to fly at high angles, allowing them to be flown in laterally constrained locations [**?**].

being on the low end.

One application of kites that concerns the carrying capacity is kite aerial photography, which consists of a camera mounted to a kite as a low-cost way to take aerial pictures [**?**]. Varying flying conditions force photographers to mount cameras to many different types of kites with differing angles of flight, payload capacity, and stability. Advanced photographers also utilize mounting rigs to stabilize and sometimes rotate the cameras in flight [**?**].

The pros and cons of a kite as a platform for the SDR system are enumerated in Table 2.4.

Table 2.4: Kite Pros and Cons. Using a kite as the aerial platform would enable a long flight time with a heavy sampling system. However, the time required to deploy the system would be high, and the maneuverability would be extremely minimal.

Pros	Cons
cost	maneuverability
flight time	deployment time
payload capacity	

2.6.2 Fixed-wing Aircraft

A fixed-wing aircraft is a rigid structure with one or more rotors oriented forward that gains lift from the flow of air over its wings [?]. A diagram showing this lift mechanism can be found in Figure 2.15. This section will focus on radio controlled (RC) aircraft since they are the type that was considered for use in this MQP. An RC aircrafts movement is typically controlled by servos onboard that move flaps to change the surface of the aircraft, in turn changing the dynamics of flight and causing the plane to respond. RC aircraft are very efficient in terms of flight time to power used because the lift comes from the shape of the wings, not the speed of the motor [?]. Since the lift comes from the wings, fixed-wing aircraft have a moderate flying time and can carry a moderate amount of weight, but the planes motion is very linear so its maneuverability is limited.

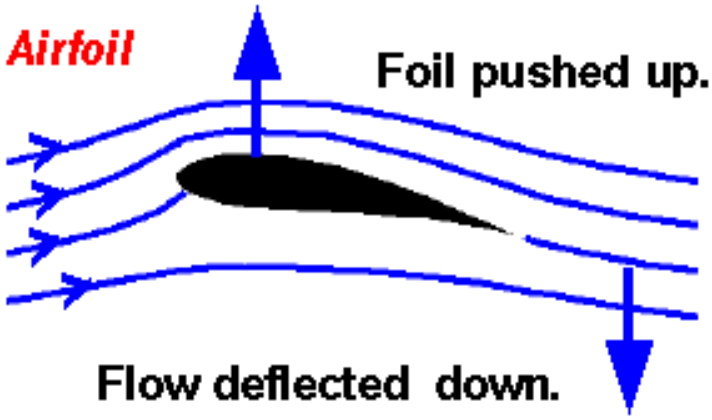


Figure 2.15: Airfoil lift generation. As air flows past the airfoil, it is forced downward, in return forcing the aircraft upward [?].

The flight time of most battery powered RC fixed-wing aircraft is around half an hour, assuming some aerial manuvering [?]. In a surveying role, with the plane flying low circles, the flight time would likely increase slightly. There are also gasoline powered aircraft with small engines onboard to power the main propeller. Gasoline

has a very high energy density, so its possible to fly for multiple hours with a gasoline powered aircraft.

Fixed-wing aircraft have a fairly lenient payload capacity. One thing to consider about adding payloads to fixed-wing aircraft is that since the aircrafts flight is so heavily impacted by the body shape, any payload will need to be incorporated into the body of the aircraft to minimize airflow disturbance. To get more lift, and thus carry more weight, the plane simply needs to fly faster to force more air over the wings. However, this has drawbacks, as spinning the propeller faster will drain the power source faster and decrease maneuverability.

Fixed-wing RC aircraft have been used in the past for similar tasks. In one example, a group of students at WPI developed a search and rescue platform using a fixed-wing RC aircraft [**?**]. An example of a fixed-wing RC aircraft is shown in Figure 2.16.

Figure 2.16: A fixed-wing RC aircraft. This is a flying wing style craft with a rear propeller. Its flight would be greatly hindered by any protrusion of the mounting system [**?**].

A fixed-wing aircraft is able to cover large distances flying laterally, but it is not able to stop mid-flight and hover in place [**?**]. Its turning radius is often quite large, and it must not drop below a certain speed to remain airborne. While fixed-wing

aircraft are mobile, they are not very agile. Most fixed-wing aircraft require a long flat runway to takeoff and land on, limiting the number of locations in which they could reasonably launch. In addition, fixed-wing aircraft are much larger and are commonly disassembled for transport.

The pros and cons of a fixed-wing aircraft as a platform for the SDR system are enumerated in Table 2.5.

Table 2.5: Fixed-Wing Aircraft Pros and Cons. Using a fixed-wing aircraft as the aerial platform would enable a long flight time, a high payload capacity, and more maneuverability than a kite. However, it would be more expensive.

Pros	Cons
flight time	maneuverability
payload capacity	cost
	deployment time

2.6.3 Multicopters

A multicopter, sometimes also called a drone, is a flying device with two or more upward oriented rotors. A typical multicopter structure will have an even number of fixed-pitch propellers (often 4, 6, or 8), powered by electric motors placed equidistant from the center of mass [**?**]. To have full control over the movement of the quadcopter, at least 3 rotors are needed [**?**]. The least mechanically complex of the three devices investigated here, multicopters rely solely on software control of the rotors for lift and control. A diagram of the forces acting upon the multicopter can be found in Figure 2.17, and an example picture of a multicopter with 6 rotors is included in Figure 2.18.

Multicopters use fixed-pitch propellers to correlate rotor speed with force produced. Lift can be controlled by uniformly increasing or decreasing rotor speed across all rotors. To effect a roll or pitch, one side's rotors are spun up and the

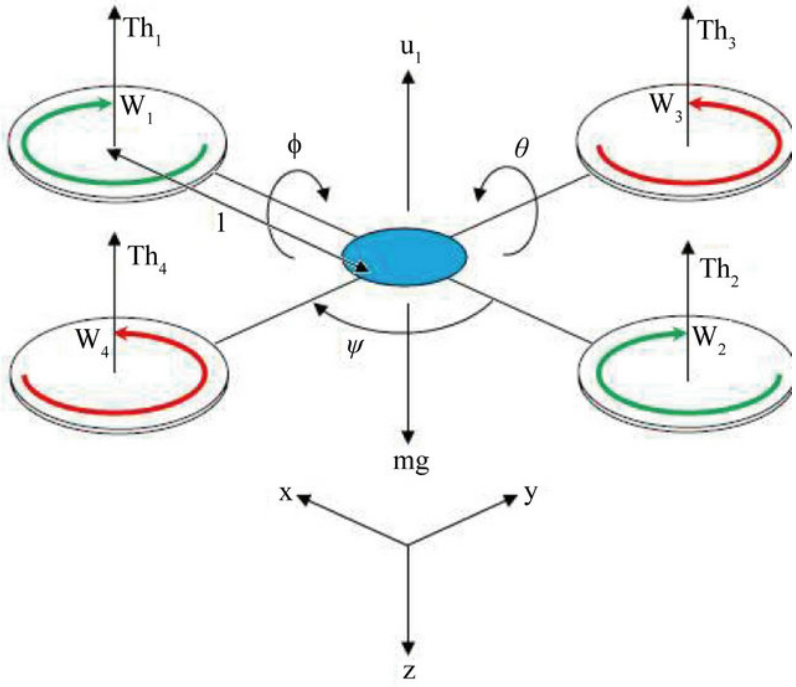


Figure 2.17: Quadcopter force diagram. Motors and propellers alternate in direction of rotation so that the drone is stable around the vertical axis. The torque generated by each propeller is canceled by the propeller opposite it [**?**].

other side's are spun down so that the net lift remains the same but the craft tilts to the side with less thrust. Multicopters also have the unique ability to yaw in place by spinning alternating rotors faster and slower. Since the rotors alternate in direction of rotation, spinning one set faster creates an imbalance of torque, which makes the whole platform rotate in place [**?**].

Multicopters are thus highly maneuverable, but at the cost of flight time and payload capacity. The primary concern when using a multicopter is the flight time. Most commercial multirotors quote flight times in the 10-20 minute range with the best performing drones topping out around 30 minutes [**?**]. This is because the power required to spin the electric motors fast enough to get lift can be immense, drawing up to 60 amps on takeoff [**?**]. Ascending from a stationary position draws much more energy than hovering or lateral flying, so multiple takeoffs and landings

Figure 2.18: A multicopter with 6 rotors. The brushless DC motors allow for very high speeds and a high degree of control [**?**].

in one flight will hamper the battery even more. The high current draw of the electric motors necessitates the use of Lithium Polymer (LiPo) batteries that can discharge large amounts of power very quickly. LiPo batteries have another benefit for use in multicopters and that is their low weight.

Weight is a very important factor in the operation of a multicopter because the amount of weight to be lifted directly corresponds to the speed the motors must be driven at to achieve lift. Multicopters in general have low payload capacities because any extra weight equates to more power that must be provided by the rotors. In effect, the more weight the drone is carrying, the less flight time it will have. This correlation forces a choice to be made over whether to prioritize flight time or amount of payload supported.

The most advantageous aspect of multicopters is their maneuverability. The nature of their control means it is possible to stop and start, hover in place, rotate in place, and quickly change directions and altitudes. In particular, the ability to

stop at a point and rotate at a fixed rate is especially unique. Deployment of a multicopter is very fast as well: there is nothing that needs assembly prior to flight, and there is no need to find a long open space to takeoff. All that is required procedurally is powering on the device and setting it on the ground, as multicopters are capable of vertical takeoff and landing (VTOL). The high maneuverability and ease of deployment make for an easy user experience, since control of a drone (with good software) is simple and very responsive. The multicopter is a platform that has very good maneuverability, but must sacrifice payload capacity and flight time.

Table 2.6: Multicopter Pros and Cons. Using a multicopter as the aerial platform enables a high degree of maneuverability and very quick deployment. However, multicopters have much lower flight time and payload capacity.

Pros	Cons
maneuverability	flight time
deployment time	cost
	payload capacity

2.7 Chapter Summary

In this chapter, background information was provided on a range of topics relevant to the undertaken project. The wireless spectrum, divided up into chunks by use, will be the medium for the analysis carried out in this project. Spectrum sensing and localization will be key to achieving the goals of the project. Software Defined Radios are useful for their flexibility around sample processing. Finally, of the flight platforms discussed, multicopters are the only ones with high maneuverability, although they sacrifice flight time and payload capacity to achieve it. This information will allow the informed discussion of implementation of the system for this project.

Chapter 3

Proposed Approach

This chapter will outline the approaches considered to accomplish this project. The goal of this project was to develop a modular platform for sensing the wireless spectrum through the use of a software defined radio (SDR) from an aerial platform. The aerial platform chosen is the multicopter drone due to its controllability and maneuverability.

The fixed aspects are as follows: the target spectrum for the project was 2.4 GHz, the software defined radio was the USRP B-200 Mini, and the antenna was the Alfa APA-M25. The software defined radio and antenna were donated to the project by Gryphon Sensors. The B-200 Mini has a small form factor or 83.3 x 50.8 x 8.4 mm, has a frequency range of 70MHz to 6GHz, is powered by USB (5V), and has an extensive set of libraries. It is an excellent choice for the spectrum sensing platform primarily due to its frequency range and small form factor. The antenna is designed to be used at 2.4 and and 5GHz, which are the two frequencies WiFi is transmitted at. The antenna is directional with a 16 degree vertical angle and a 66 degree horizontal angle. The directionality will allow for more accurate determination of the source of the WiFi signal.

3.1 Prototype Systems

A multicopter was chosen to be the aerial platform due to the increased maneuverability provided. Theismaneuverability will allow the platform to perform well in any target environment chosen for spectrum sensing. However, with the drones maneuverability, it sacrifices vital airtime and payload capacity, creating design constraints that will be considered for each of the following approaches. The brainstorming process led to four distinct design concepts that utilized a multicopter for spectrum sensing.

It should be noted that these four designs incorporate similar devices for the single board computer, storage, battery and GPS. Regardless of which drone approach was taken, an onboard processor and storage device had to be chosen for the processing and storage of IQ data. The single board was selected from five options which are shown in Appendix A.1. The final decision was to use the UP board. The decision was primarily between the UP board and the ODROID-XU4. Both of these boards have a similar form factor, but the UP board draws 1A less current. This reduces the power consumption of the board considerably and negates the lower weight of the XU4. An additional reason for choosing the UP board was the shipping location. The UP board was shipped from Europe, while the XU4 was shipped from Korea. Therefore, the lead time for the UP board would be shorter and documentation more accessible. As for storage, the selected device was a 32GB USB flash drive, which is capable of storing all the data gathered in one flight simulation of 20 minutes.

The external GPS and the battery were chosen for their small form factor. The GPS chosen was the GlobalSat ND-105C micro USB GPS receiver which has a form factor of 30.4mm x 15.4mm x 4.5mm. The battery chosen was the 11.1 Volt

Lumenier 800mAh 3s 20c LiPo battery which had a form factor of 55mm x 31mm x 20mm. This battery, when stepped down to the voltage used by the UPboard, provides enough power for the computer to run 20 minutes at maximum power draw, the same amount of time as the drone can fly.

The first approach concept is based upon having an individual wireless communications frequency for each task. Initially this design was centered around the Autel X-Star that was controlled at 5.7 GHz band. However, upon learning that the 3DR Solo was configurable to operate at 5.7 GHz, and based on how well the 3DR Solo is documented, the approach was modified to center around the 3DR Solo. The 3DR Solo is a drone designed as a consumer drone used for aerial photography. The camera and gimbal were replaced with the modular spectrum sensing system to better utilize the limited payload capacity of this drone. This drone has a payload capacity of 700 grams, which allows for a substantial payload. While the drone has a large payload capacity, the design will attempt to minimize the payload weight and enable the platform to be used on a larger number of drones. Furthermore, this drone can be configured to operate with the controller at 5.7 GHz that does not interfere with spectrum sensing at the 2.4 GHz wireless band. The transmissions back to the user would be on a frequency of 900 Mhz if real time response is required [**?**]. More detailed design configurations on the first approach can be found in Appendix A.2.

The second approach concept is focused on assuring that this project is feasible. This assurance is from the fact that the DJI S1000 octocopter was designed to carry heavier payloads. The DJI S1000 was designed for professional photography and cinematography, where stabilization and high payload is essential. As a result of these qualities, as well as the more premium target market, this drone is priced higher than many consumer-level drones. This drone is able to carry payloads of up

to 2 kilograms, providing large flexibility in terms of design constraints. However, because of the flexibility this high payload offers, the resulting design may not be able to be used on consumer drones with a lower payload capacity. Another advantage this drone presents is that the communication between the controller and the drone is operated at 900 MHz, which will leave the detection of the 2.4 GHz wireless band unimpaired. This approach serves better as a backup plan in case the modular system can not be lifted with the commercial drones used in testing. More detailed design configurations on the second approach can be found in Appendix A.3.

The third approach is based on using an autonomous navigation approach that was considered with the 3DR Solo drone. One of the features that was provided with the drone is autonomous flight planning, where the user can set way points for the drone to fly. This feature will allow simplistic controls, however, general drones do not have this feature implemented. Furthermore, the 3DR Solo drones ability to be customized is valued, as the initial controller-to-drone transmission frequency is in the target sensing frequency of 2.4 GHz. The autonomy of this design eliminates the need for constant communications with the drone while flying, preventing the control signals from interfering with the spectrum. More detailed design configurations on the third approach can be found in Appendix A.4.

The fourth approach is focused on shielding the drones control antenna from the spectrum sensing antenna. This configuration was created to ensure that the wireless control signals received on the drone and the wireless spectrum being detected would not interfere with each other, since they are on the same frequency. Therefore, a drone carrying 2 antenna configurations would be needed- one that would communicate with the ground, and the other that would perform spectrum sensing. The antennas would need to be isolated from each other in order to ensure

the 2 antenna configuration works. This approach would help add resilience in the drone platform and would allow for flexibility on communication protocols from the ground to the drone. The primary disadvantage to this approach is the additional weight that the shielding would add to the design. More detailed design configurations on the fourth approach can be found in Appendix A.5. The table below compares and contrasts the four approaches.

Table 3.1: Pros and cons of the four approaches.

Approach	Pros	Cons
1	No interference	Minimal payload capacity
2	Great payload capacity & No interference	Not a commercial drone
3	Does not require communication signals	Flight path has to be predetermined
4	N/A	Increased payload for sheilding

3.2 Project Planning

Microsoft Project was used to plan out objectives throughout the project. The Gantt chart in Figure 3.1 shows the project objectives. The planning came in three parts: implementation, testing and writing the report. In the initial stages of implementation, each individual part was done first. When each part worked separately, the blocks were then integrated to make sure that the system functions as planned. From then on, the project development moved to the testing stage, where the system was tested in three scenarios. These scenarios allowed ease of debugging if the testing does not go according to plan. The SDR was tested to determine whether it can locate a WiFi signal without any movement. Then, before mounting the system on the drone, the system was moved around to check whether the data correlates to a specific direction of a transmitter when the system is mobile.

Finally, when both tests worked properly, the system was mounted on the drone and tested while flying. During the process of implementation and testing, the report was be written based on the work done in the implementation and testing stages.

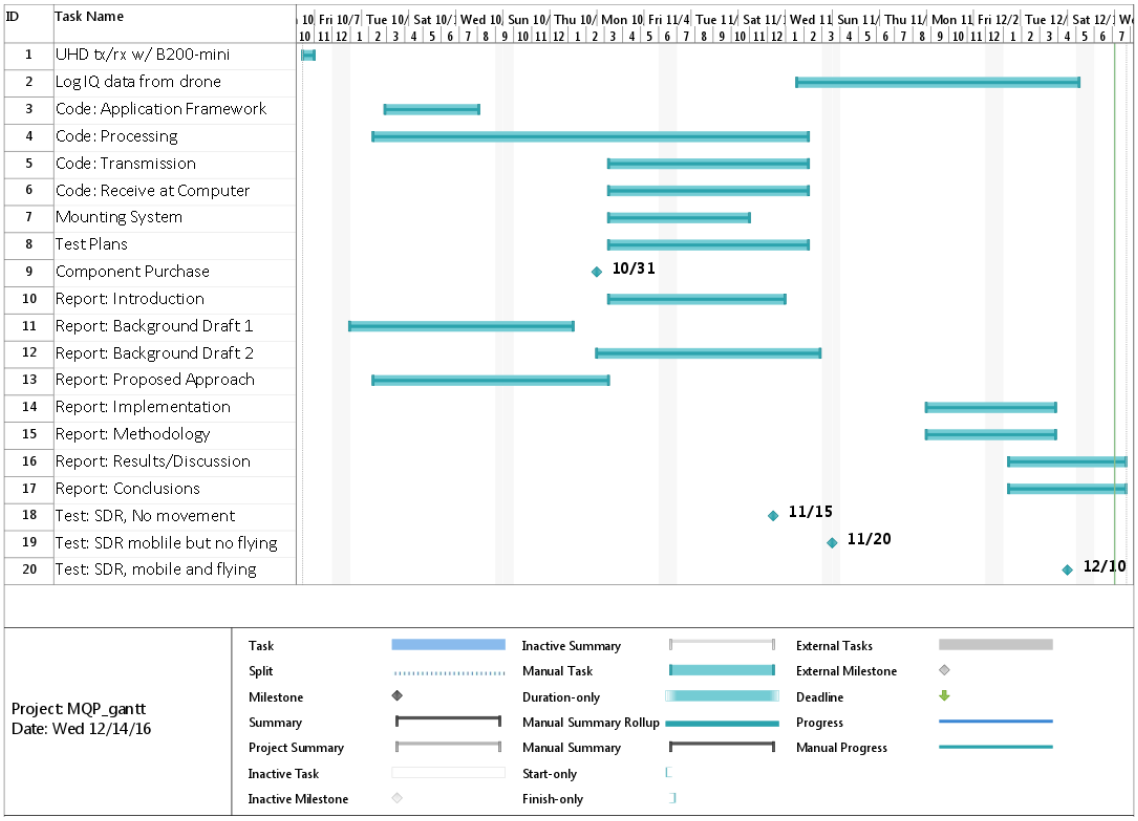


Figure 3.1: Project planning gantt chart. This chart was used to display deadlines and ensure that tasks were being completed. Team members were assigned tasks and were held accountable for those tasks.

Chapter 4

Implementation

This chapter details the implementation of the MASDR platform. The description has been divided into software and hardware for clarity.

4.1 Software Implementation

The software for the MASDR system was written in C++ and can be found in Appendix **??**. Documentation was done with doxygen, allowing for dynamic updates to the docs whenever changes were made. Additionally, some scripts were written to assist in other aspects of the project.

During the design phase, there was discussion over whether to use GNU radio or C++ for the MASDR application. GNU radio would allow for more complex signal processing patterns to be constructed more easily in comparison to C++, but would increase the overhead on development and potentially at runtime as well. Using C++, the application would have to be coded entirely from scratch, interfacing with the USRP Hardware Driver (UHD) libraries, but would have much lower overhead. Based on the skills of the team members, as well as the desire for the application to be as low power and simple as possible, C++ was chosen for the

application.

The MASDR Application is structured as a central C++ class with a few additional utility functions external to the class. The class contains methods for initialization, sampling, processing, and transmission among others. Certain services, notably updating the GPS and taking in samples, are performed in separate threads to allow for the parallelization of tasks. In the applications main loop, an instance of the MASDR class updates its status and begins or changes its task based on the new status. Status is composed of both physical status, namely location, and software status, representing the current processing state. The physical status structure includes a member variable for heading, although this is unused in the current version of the platform. The processing state is an enumerated type with values PROCESS, TRANSMIT, and IDLE. The state of the device determines what the processor will be doing. During operation, the application will flow from state to state as described in the state transition diagram in Figure 4.1.

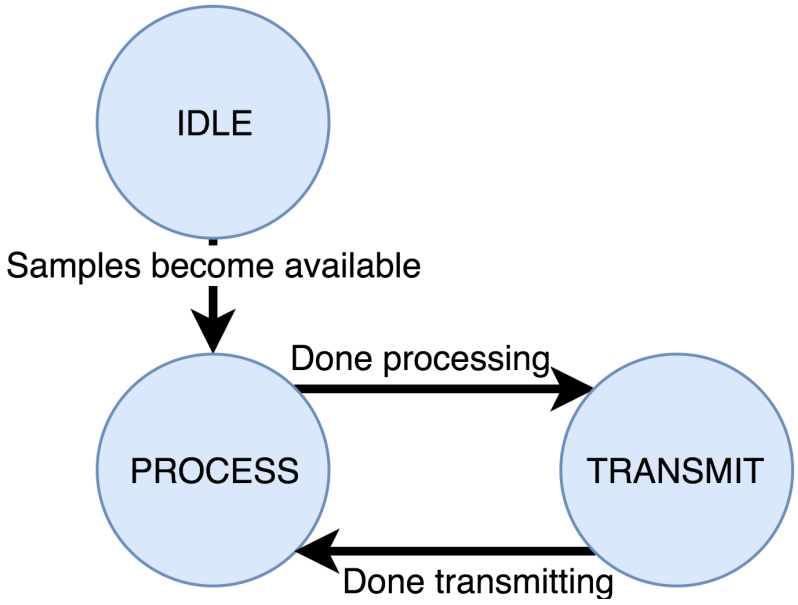


Figure 4.1: MASDR State Diagram. Software states change according to the graph described in the image.

The onboard software is the majority of the project, however there are other essential pieces. On the ground station side, a GNU radio receiver intercepts and decodes all transmissions from the aerial platform. Additionally, a mapping tool was created to calculate the approximate distance to signals and draw circles on a map to visually locate signals. The mapping tool consists of a Python script that performs the distance calculations and populates a Javascript list of GPS locations and distances. The Javascript calls into the Google Maps API to display the distance rings on a map. Finally, a Python script was written to command the drone to autonomously spin 360 degrees.

While the original plan was for the sampling routine to be driven by the motion of the platform, a lack of precision and accuracy in the GPS readings rendered this nearly impossible. Therefore, the SDR was configured to constantly take samples, with control originating from a separate processing thread from the main thread. The use of threads allows for shared data throughout the program, which is used to share the buffer of samples with the main thread.

In order to avoid any shared data bugs, the sampling routine fills a buffer of blocks one at a time, eventually overwriting the oldest block to record each new block. At the beginning of the processing routine, a copy of the most recent half of the sample blocks is made. This makes it impossible for the sampling thread to overwrite a block of samples that the processing thread is using.

There is a defined motion pattern that the quadcopter should follow in order achieve correct results. At a constant altitude, the quadcopter should travel to a series of points, stopping and rotating 360 degrees at each point. The constant altitude will allow for more accurate localization of signals based on RSSI distance.

The samples taken by the SDR are then processed to extract useful information. Given that the MASDR system is designed to be low power, a delineation between

onboard processing and processing at the ground station was created. The UP boards processor is fairly capable, allowing the initial steps of the signal processing to be done onboard. A more indepth description of the processing done onboard the MASDR platform can be found in Section 5.1. The information is processed as much as possible onboard to reduce the size and duration of the transmissions to the ground station. The initial samples are also stored on a flash drive onboard so that post processing can be done on the raw data.

To detect the existence of a signal, an OFDM receiver in the MASDR application looks for beacon signals in the sampled data, block by block. The RSS of a beacon signal is recorded with its corresponding location, the location the drone was at when the block of samples were taken. This mapping is then transmitted down to the ground station for further processing. The MASDR platform is designed to transmit data between processing blocks of samples. The transmission protocol consists of two types of messages: a header packet, and a data packet. The header packet is the first packet sent after processing the data; it contains a transmission ID, the location of the sampling block, and the number of following data transmissions. The data packet consists of a transmission ID which corresponds to its header packet, a heading, and a signal strength for beacon signals. Again, the heading is unused in the current version of the platform, but a future version could make use of the field.

4.2 Hardware Implementation

The hardware for the MASDR system consists of the components used and the mounting system employed to attach it to the drone. Using the 3DR Solos development guide as a reference, the mounting system was designed to hold all the components securely to the drone. By using the official mounting points on the

drone, the hardware was made as robust and efficient as possible. In order for the hardware to be mounted properly, the MASDR system was mounted onto the drone platform by using the 3DR Solos preexisting hardware mount. To do this, the 3DR Solo developer's guide was used to determine the dimensions of the mounting screws so that the mounting platform was able to be easily screwed onto the 3DR Solo [**?**]. The mounting system connects to the bottom of the 3DR Solo with four M2 screws that are spaced in a 63mm x 41mm rectangular pattern as seen in Figure 4.2.

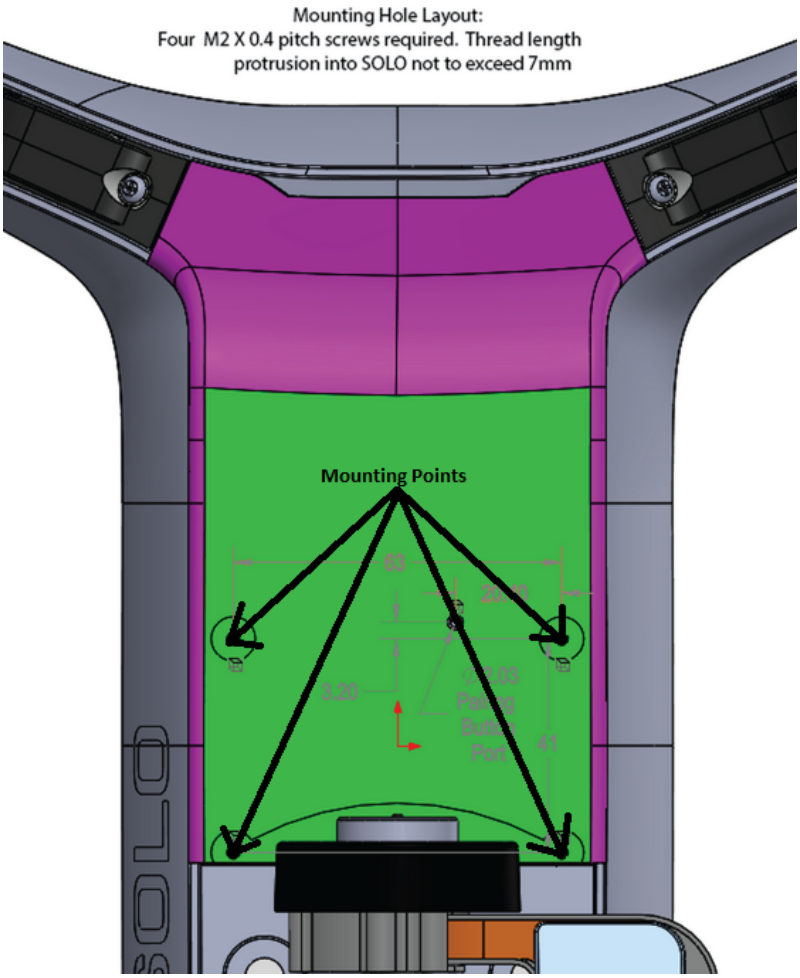


Figure 4.2: 3DR Solo Mounting Points used for attaching the modular system.

The mounting system consisted of two pieces of wood laser cut with and mounting pattern, with unneeded wood removed to reduce weight. The pieces were at-

tached with 3 inch standoffs, allowing room for the UP board, B-200 mini, tranformer, and battery to be mounted and connected together. A picture of the assembled mounting system can be seen in Figure 6.2 and Figure 6.4.

In order to ensure that the MASDR platform is powered for the entire flight, a battery was needed that could hold as much energy as possible while not adding too much weight to the drones payload. In addition to this, the battery needed be able to supply as much current as the hardware could draw. This became an issue for finding a suitable 5V battery that would support a maximum current draw of 4 Amps. Instead of a 5 volt battery, a 12 volt battery was used, and a DC-DC converter was used to match the 5V 4A requirement. The chosen battery was an 11.1 volt lithium-polymer battery weighing 66 grams. It was then connected to a DROK DC voltage converter that both transformed and regulated the voltage to 5V. By downconverting the voltage from the battery, the system is able to draw more current from the battery than what is usually supported by a 5 volt battery.

The 3DR Solos default wireless card communicates to the controller using the 2.4 GHz band. This interferes with sensing and locating signals on the 2.4 GHz band. In order to avoid this issue, the 2.4 GHz network card of the 3DR Solo was replaced with a dual band 2.4 GHz and 5 GHz network card. Unfortunately, even with the new wireless card, the controller and drone were unable to communicate on the 5 GHz band. It is suspected that this is due to incompatible antennas or configuration scripts on the controller.

4.3 Implementation Summary

In this chapter, the implementation of of the MASDR system was detailed from both the software and hardware points of view. The software description covered

the application structure, sampling procedures, onboard processing strategy, and data transmission protocol. The hardware section described the mounting system, power conversion, and networking card replacements.The implementation of the system went through multiple iterations during development, and further tweaking and additions to the implementation of the platform are encouraged as needed for any work done using the platform in the future.

Chapter 5

Methodology

5.1 Spectrum Sensing

The main goal of this system is to be able to detect wireless signals. This is done in with a combination of energy detection, for determining if there is any signal in the channel, and a matched filter in the frequency domain, to determine if there is a WiFi signal present. Each will be described in the following section.

The system detects transmissions based off of energy detection. This type of detection is done by measuring our received signals strength along a desired central frequency, and comparing it to an ideal threshold. In order to determine the energy of the received signal, the following equation was used:

$$E = \int_{-\infty}^{\infty} |x(t)|^2 dt$$

$x(t)$ is the received signal, bandpass filtered to only use the bandwidth of WiFi. This energy measurement is then compared to a threshold energy of -85 dBm. This threshold was chosen due to it being high enough to ensure a low chance of false alarm while still sensing signals from a significant distance, which is necessary since

the drone was high in the air.

In order to categorize WiFi signals at the 2.4GHz band, an OFDM detector was implemented. In order to detect OFDM, the receiver will detect beacon frames by correlating upon the WiFis L-STF structure. The L-STF structure is a specific spacing and timing of transmissions over various subcarriers. This structure is also used for calculating the coarse frequency offset. The subcarriers and transmissions used in L-STF are shown in Figure 5.1.

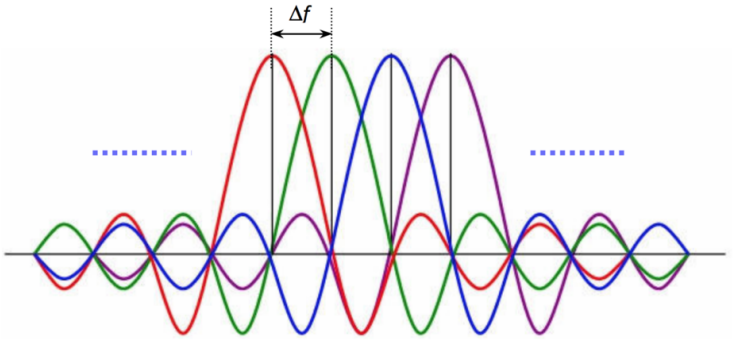


Figure 5.1: This photo gives an example of what OFDM subcarriers look like when displayed upon an FFT. Each subcarrier is shown as its own respective pulse and the subbcarier frequency offset is denoted by Δ f.

The multiple subcarriers are represented by the various waveform colors in Figure 5.1. The spacing of each frequency is denoted by f. This series of waveforms are transmitted from the OFDM transmitter over a fixed amount of time. The spacing and frequency behavior of the L-STF field has characteristics defined in Table 5.1:

Once a beacon transmission is detected using the above techniques, the L-SIG field must be decoded. The rate field gives us information such as the modulation, coding rate, and data rate by relating the rate to its respective binary value. This field is described in the table below.

By the rate information, the received packet's 802.11 frame can be further analyzed to find the broadcast SSID of our received transmission frame. The SSID

Table 5.1: This table gives an outline on the L-STF Characteristics for an OFDM transmission. Due to our Wi-Fi signals being in the 20, 40, and 80 MHz range, the OFDM match filter will use the values in the first row in order to perform spectrum sensing. The match filter will filter upon a subcarrier spacing of 312.5 kHz by noticing a change in the center frequency of a transmission.

Channel Bandwidth (MHz)	Subcarrier Frequency Spacing, Δ_F (kHz)	Fast Fourier Transform Period ($T_{FFT} = 1/\Delta_F$)	L-STF duration ($T_{SHORT} = 10 * T_{FFT}/4$)
20,40,80,160	312.5	$3.2\mu s$	8 μs
10	156.25	$6.4\mu s$	16 μs
5	78.125	$12.8\mu s$	32 μs

Table 5.2: This table gives the corresponding modulation, coding rate, and data rate based on the received binary information from the L-STG field. These fields can then be used to further demodulate the received signal.

Rate(bits 0-3)	Modulation	Coding Rate (R)	20 MHz data rate (Mb/s)
1101	BPSK	1/2	6
1111	BPSK	3/4	9
0101	QPSK	1/2	12
0111	QPSK	3/4	18
1001	16-QAM	1/2	24
1011	16-QAM	3/4	36
0001	64-QAM	2/3	48
0011	64-QAM	3/4	54

field which starts at the 36^{th} byte of the beacon frame header is the only desired information here as it allows the identification of separate access points.

By converting these bits to ASCII, the SSID of the received transmission can be obtained.

Using a combination of energy detection and match filtering on OFDM beacon frames, the MASDR system was able to identify the desired signals with a high probability of detection and a low probability of false alarm. Additionally, this makes accurate determination of the received SSID possible.

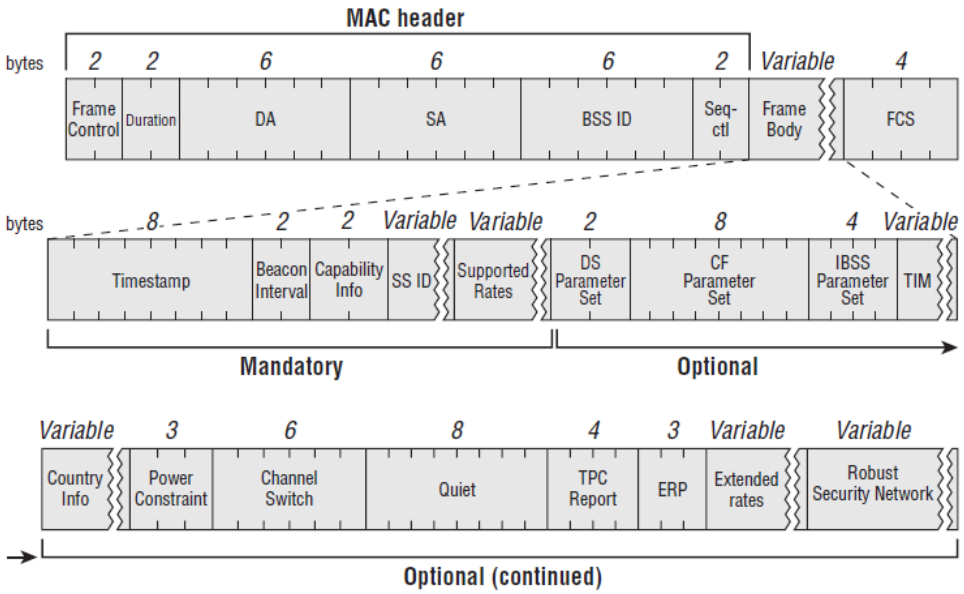


Figure 5.2: MAC level structure of beacon frames [**?**].

5.2 Fusion and Localization

In order to draw meaning from the spectrum being measured, two types of data need to be analyzed: received signal strength (RSS) and location. Both of these measurements have noise to some extent. In this section, the combination of these measurements and the mitigation of sensor noise was discussed.

After identifying a signal, the next step is to locate the signal source. Out of the methods described in the background, most are ruled out based on the non-cooperative nature of the sensing, leaving only Angle of Arrival (AoA) and RSS localization. Of those two methods, Angle of Arrival is the significantly more complex method to implement, requiring either a strongly directional antenna or an array of antennas [**?**]. As such, RSS localization was the method implemented. This method requires three stationary observation points, as opposed to the single moving observation point contained on the aerial platform. To get around this requirement, it is assumed that the device broadcasting the WiFi is stationary [**?**]. This assump-

tion is valid because most access points are indeed stationary and the drone moves quickly enough that even a slowly moving signal would likely be able to be roughly located. This scenario is illustrated in Figure 5.3.

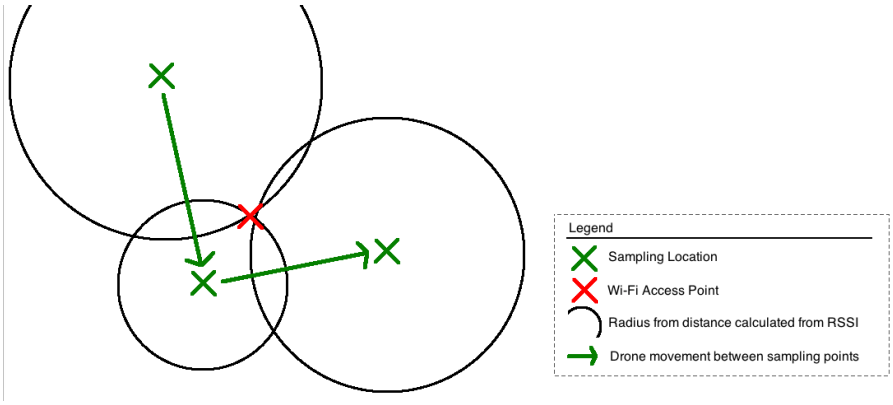


Figure 5.3: Diagram of triangulation with drone movement. In a traditional triangulation scheme, samples are taken at all three locations simultaneously. In the situation described here, since the Wi-Fi access point is unmoving, the drone is able to move between sampling points and take samples sequentially instead of simultaneously.

Occasionally while sampling, the drone will rotate continuously in a process described in Section 4.1. This presents a complication to the processing of the signal: only one sample is recorded at a given angle. Since the samples must be processed in blocks, each block will represent samples from a range of angles. Since the heading measurement is not implemented in the current version of the system, this is of little concern, but for future use, the scenario has been analyzed.

The angles included in a block are determined by the amount the drone rotates between the first sample and the last sample of the block. In a small enough block, the error generated by the difference in direction is minimal. However, a smaller block represents a smaller amount of time, in which the signal is more prone to being affected by noise. One solution is to slow down the angular velocity of the drone. While this can be done, the resulting design would be less modular, depending more heavily on the programmability of the drone in use. The other solution is to

increase the block size, increasing the amount of time measured, but introducing the problem of deciding what direction should correspond with the resulting signal measurement. Regardless of how this is chosen, the larger block size increases the error in the localization measurement.

In a magnetometer-enabled version of the system, a block size of 131,072 should be chosen. The 3DR Solo rotates at around 2 seconds per rotation. This means that it takes roughly 0.0222 seconds to rotate 4 degrees. At a sample rate of 5 MSPS, this is 111,111 samples. In order to input the signal into the FFT, the block size has to be a power of 2. The nearest power of 2 is 2^{17}, which is 131072. Each block consists of the second half of the previous block and the next half block of samples. The first block has no previous block, so is just the first 16384 samples. The processed measurement is associated with the direction in the middle of the angle sweep covered by the block. This method introduces a maximum of one degree of error, which is on the same order of magnitude as the error in the HMC5883L magnetometer, a common chip [**?**].

The shifting nature of wireless channels introduces a difficult to predict noise into the measurements taken. In order to mitigate this, the drone can rotate multiple times. The resulting received signal strength measurements can then be averaged with the readings from each rotation. This averaging reduces the chance that a random shift in the channel will negatively impact the direction readings.

The MASDR system uses GPS readings for location. The distance of the beacon from the drone at multiple points is then used to get an estimated location of the beacon. Within a GPS receiver, signal correction is often carried out. This results in a cleaner and usually more accurate measurement. However, the GPS receiver that is used in this project provides very noisy results, providing at best 25 meter accuracy when tested in one location. A contributing factor to this is that the

receiver used is both small and cheap. Consequently, the values that it reads are frequently very scattered, with a high variance. In order to mitigate this, a four-state Kalman Filter was designed and simulated in MATLAB. The Kalman Filter equations presented in the Chapter 2 are reproduced below [**?**]:

Predict:

$$\vec{x} = \mathbf{F}\vec{x} + \mathbf{H}\vec{u} \tag{5.1}$$

$$\mathbf{P} = \mathbf{FPF^T} + \mathbf{Q} \tag{5.2}$$

Update:

$$\vec{y} = \vec{M} - \mathbf{H}\vec{x} \tag{5.3}$$

$$\mathbf{S} = \mathbf{HPH^T} + \mathbf{RS} \tag{5.4}$$

$$\mathbf{K} = \mathbf{PH^T S^{-1} K} \tag{5.5}$$

$$\vec{x} = \vec{x} + \mathbf{K}\vec{y} \tag{5.6}$$

$$\mathbf{P} = (\mathbf{I} - \mathbf{KH})\mathbf{P} \tag{5.7}$$

The state $\vec{x}$ holds the x position, y position, x velocity and y velocity.

$$\vec{x} = \begin{bmatrix} d_x \\ d_y \\ v_x \\ v_y \end{bmatrix} \tag{5.8}$$

Since we are just estimating the GPS measurement, there is no input that adjusts

the values, so $\vec{u} = 0$. The state transition matrix $\mathbf{F}$ is based on how elements of the states interact with each other. In order to ensure that the filter isn't too complex, a constant velocity model is used [**?**]. With a small enough dt, this is a valid assumption. Because velocity is the derivative of position, $\mathbf{F}$ is defined as follows:

$$\mathbf{F} = \begin{bmatrix} 1 & 0 & dt & 0 \\ 0 & 1 & 0 & dt \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{5.9}$$

$\mathbf{P}$ is the variance matrix for the state, and is initialized as a diagonal matrix with the variance of each individual element. While the assumption that position and velocity are independent is oversimplified and naïve. However, since $\mathbf{P}$ gets updated, this assumption is acceptable. $\vec{y}$ is the difference between the most recent measured value $\vec{M}$ and $\mathbf{H}\vec{x}$. H is set as an identity matrix, so $\vec{y}$ ends up being the difference between a measurement and its prediction. Both $\mathbf{Q}$ and $\mathbf{R}$ are noise covariance matrices; $\mathbf{Q}$ is the environment noise covariance, and $\mathbf{R}$ is the model noise covariance. In order to avoid complications, both of these are assumed to be piecewise white noise. this produces the matrix:

$$Var_v * \begin{bmatrix} \frac{dt^4}{4} & 0 & \frac{dt^3}{2} & 0 \\ 0 & \frac{dt^4}{4} & 0 & \frac{dt^3}{2} \\ \frac{dt^3}{2} & 0 & dt^2 & 0 \\ 0 & \frac{dt^3}{2} & 0 & dt^2 \end{bmatrix} \tag{5.10}$$

The variance is chosen to best fit the data.

K is the Kalman gain of the system. With a constant noise and variance, this matrix is constant. This is beneficial, as the matrix inverse (or pseudo inverse) that is needed to compute **K**. In this implementation, the MATLAB simulation will approximate **K**, and the resulting values was used.

Equations (5.6) and (5.7) are used to update the state and state covariance with the most recent measurement.

Both of the two measured values play a role in the localization of a beacon. The GPS location provides a base location from which to reason about the location of the beacon. This location is produced from a Kalman filter, reducing the variance of the measurements. Then, using the RSS measurement of the signal, a ring of possible locations is found around the GPS location. Eventually, measurements from multiple points are used together to get an accurate location of the beacon. Both of these values also have some noise inherent to the process that they are measuring. By taking multiple measurements and calculating the intersections of the rings to signals, the accuracy of the localization is improved.

5.3 Drone Control and Communications

When dealing with multiple frequencies, interference is an issue that must be considered. Interference in the scope of this project may lead to interruptions to drone control, faulty measurement of data, and faulty transmission of data. To help solve these issues, the following steps were taken in drone control and drone communications.

With the choice of using remote controlled drones to sense WiFi signals comes the problem of interference between the control signal and the WiFi being searched

for. This problem is relevant with the use of the 3DR Solo, as the chosen WiFi detection frequency is 2.4 GHz and the 3DR Solos operating frequency is also at 2.4 GHz. To alleviate the possible skew of data that the control signal may cause, the 2.4 GHz wireless card of the drone and the controller was changed to a dual band card that supports the 5.7 GHz band. Real time data transmission from the aerial platform to the ground station uses the 900 MHz band. Between the two additional bands used, the 5.7 GHz band was chosen to be the control frequency because of its shorter range due to higher frequencies losing energy faster. The logic behind the decision is that if the drone cannot be controlled properly, then the data obtained from the transmission would not be of significance. Furthermore if the control frequency was 900 MHz, then the drone would be controllable farther away than it would be able to transmit data back to the user, which is not a useful feature.

With the popularity of the 3DR Solo and its ability to be customized, an online discussion board was created for 3DR Solo users to share their experiences with other fellow users [**?**]. This discussion board provides relevant information on the capabilities and drivers of this commercial drone and customizations that different users have tried. In one thread, a user significantly improved the controls communication range of the drone by swapping the 2.4 GHz wireless card to a higher powered one. This example was the basis of changing the wireless card from 2.4 GHz to 5.7 GHz.

The wireless cards used on the 3DR Solo are mini PCIe and are based on the Atheros AR9382 chip [**?**]. Both the controller and the 3DR Solo have these cards for transmission and receiving. The 3DR Solo also uses the ath9k driver that works with a range of Atheros based cards. Based off these requirements, the possible wireless cards that were suitable for replacing the ones on the 3DR Solo were limited. The

decision was made to use the MikroTik R11e-5HnD card because it uses the Atheros AR9382 chip that is compatible with the 3DR Solo. As complications arose, detailed in Section 6.2, the SparkLAN WPEA-121N MiniPCI-E Half-Size wireless card was used instead. When physically replacing the wireless card, a YouTube tutorial was used for guidance [**?**].

For the drone to send data back to the user, the wireless carrier of 900 MHz was used so that it would not interfere with either the control frequency or the detection frequency. The data packet starts off with multiple consecutive start bits, signaling to the user the beginning of the data transmission. Following the start bits is the data packet itself, of which there are multiple different types, described in Section 4.1. The data inside the packet is interleaved 5 times, meaning that the data will recur at least 5 times to ensure that the data received at the receiver side is consistent. The modulation scheme is DBPSK to increase the quality of the received signal. Furthermore, to avoid corruption of the data packet, a checksum is calculated and appended at the end of the packet. The transmission is concluded by a stream of zeros. Additionally, to make it easier to decode the signal, the signal is modulated with a root raised cosine (RRC) filter.

Figure 5.4: A state diagram summarizing how the drone communicates.

5.4 Testing

For this project, two main categories of test environments were selected for testing the aerial software defined radio: rural and urban areas. The initial testing was in the rural areas, where there are limited interfering wireless signals. The testing was done in a forested backyard so that the SDR was isolated from any interfering

communications, providing a good model of using the drone for search and rescue tasks. The path of a signal is shown below in Figure 5.5

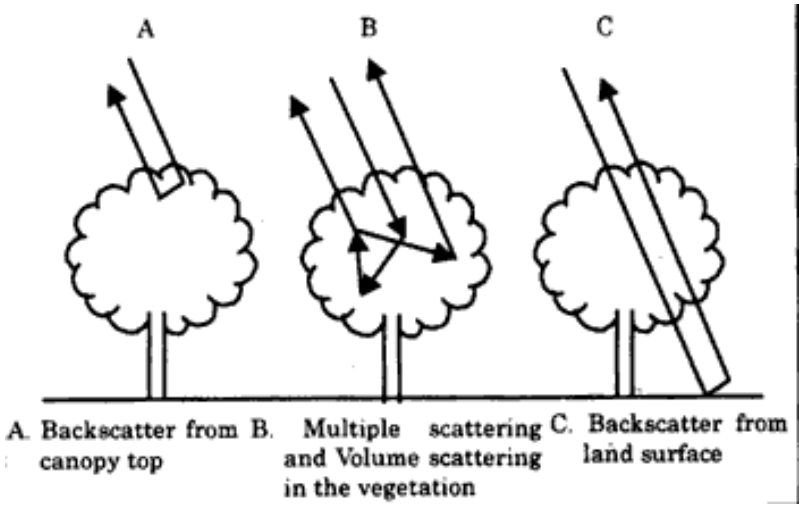


Figure 5.5: The possible paths of a signal through vegetation [**?**].

When testing in urban areas, the drone was used to collect IQ data to map hotspots and eventually locate a specific wireless signal. However, testing in urban areas will require more planning due to air restrictions in cities as the drone can be an issue regarding privacy and possible physical collision. Urban areas are also more likely to have conflicting signals using the 2.4 GHz frequency band. The team will collaborate with Worcester Polytechnic Institute (WPI) to use their parking area in the city, shown below in Figure 5.6 to test out the full system.

The first test was to ensure that the SDR is functioning properly. In this test, simple transmitter and receiver code was loaded onto the board. The SDR will then transmit a signal which was acknowledged and logged by a receiver. The receiver will then send out a signal that the SDR will receive and log. The logged signals was observed and compared to the expected result. The test plan is shown below:

1. Turn on receiver

2. Turn on SDR

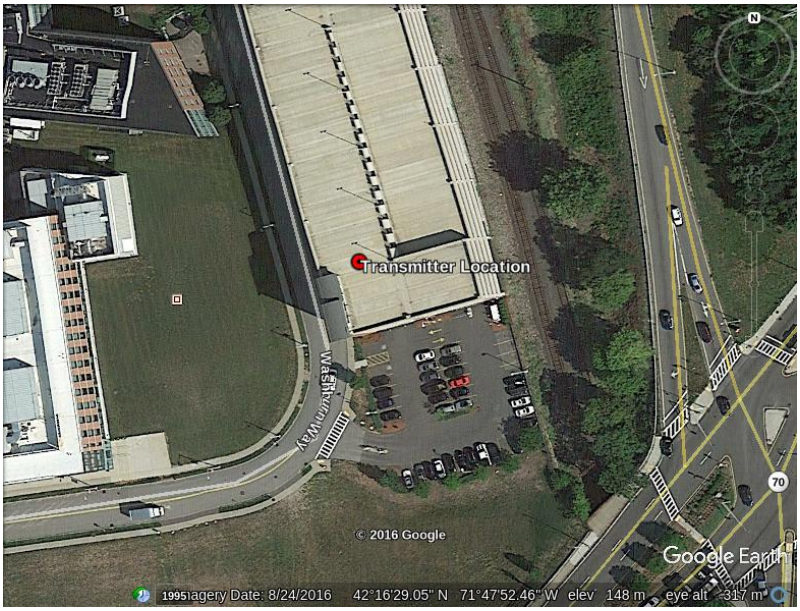


Figure 5.6: Location of the Gateway Parking Garage where the urban test was conducted.

3. SDR transmits signal to receiver

4. Receiver sends acknowledgement to SDR

5. SDR logs acknowledgement

The following tests waere conducted using the SDR and a controller as a WiFi transmitter in a parking lot, in a wooded area, and an urban area. The parking lot provided an environment that has a large open space to ensure that the SDR and the WiFi transmitter maintain an unobstructed line of sight. A parking lot with minimal WiFi interference was used to allow for near ideal conditions for preliminary testing. The rural environment provided no interference from other WiFi sources, but provided obstructions in the form of foliage. The urban area provided an environment with both obstructions and non-line-of-sight. This environment was the most challenging to locate the WiFi transmitter in. The following tests were conducted in each of these areas with the SDR not attached to the drone and attached

to the drone. When the SDR is attached to the drone, the tests were repeated at different heights, increasing by 30 feet with the maximum height being 150 feet.

The first test will consisted of the SDR being stationary and the WiFi transmitter moving. The test began with the transmitter being placed directly in front of the SDR. The transmitter was moved in a direction, first straight out from the antenna, then to the side of the antenna, and final to the back of the antenna. This was done at a constant speed and the received signal strength was measured over a period of 30 seconds.

The next test was to move the SDR with a stationary transmitter within the line of sight of the SDR. The SDR was moved to different points around the transmitter while measuring the received signal strength. At each point, the SDR stopped and rotated 360 degrees at a steady pace with the rotation lasting approximately two and a half seconds. In the urban and rural environments, this test was also run with the transmitter out of the line of sight of the SDR. In the rural area, this was behind a tree and for the urban area behind a car or building. These tests was performed multiple times and the data was analyzed to determine whether the drone was able to locate the transmitter. The test steps are shown below:

1. Position transmitter in a known location (in or out of LOS)
2. Turn on drone and controller
3. Fly drone to the testing altitutde
4. Fly drone in a grid pattern
5. Approximately every 20 feet rotate the drone 360 degrees at a constant speed for 2.5 seconds
6. Repeat step 5 until grid pattern is complete

These tests verified that the system is working correctly. Each test increases in complexity and exercised the system to observe its performance in different environments. The first test is simple to verify that the components are working correctly. The second test is done to measure the signal strength at different distances and angles relative to the antenna to be able to evaluate the range at which a transmitter can be detected. The final tests was to localize the signals and distinguish between multiple transmitters.

5.5 Chapter Summary

In this chapter, details about the methods used in this project are discussed in detail. Energy detection and matched filtering are used for identifying if there is WiFi within the area. Once confirmed that there is a WiFi transmission within the vicinity, localization occurs using the RSS method which is based upon the drone's GPS location. As a trade-off for the small size and cost-effectiveness of the GPS chosen, a Kalman filter was designed to increase accuracy of the measurements to project a better localization map of the area. Another major consideration was determined as the communicating control messages to the drone and the information messages to the ground as these two must not interfere with the WiFi sensing. In addition, the different test protocols for the two different target regions were detailed to determine the feasibility of the modular platform.

Chapter 6

Experimental Results

In this chapter, the results of the project will be discussed. This chapter is split up into System Integration, Drone Control, Spectrum Sensing, Spectrum Localization, and GPS sections.

6.1 System Integration

This section details the physical system and the integration of the hardware. This includes power consumption, system layout, and component failures or difficulties. The system diagram shown in Figure 6.1 will be explained in detail in this section.

The encasing for the system was designed in SOLIDWORKS. It was fabricated using wood panels and a laser cutter. The encasing was designed to be as minimal as possible to reduce the weight of the system. This was done by cutting holes in the panels to remove as much material as possible and using metal supports instead of wooden walls. The metal supports were used in each of the corners to connect the two wooden panels as seen in Figure 6.3.

The two wooden panels were created identically to reduce design time and fabrication time. Screw holes were precut into the wood to ensure mounting the com-

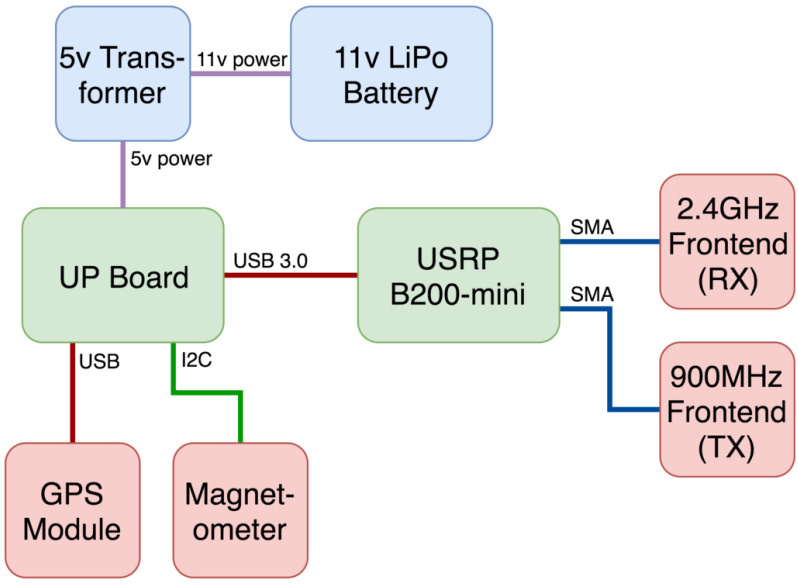


Figure 6.1: MASDR system diagram detailing the connections between each component.

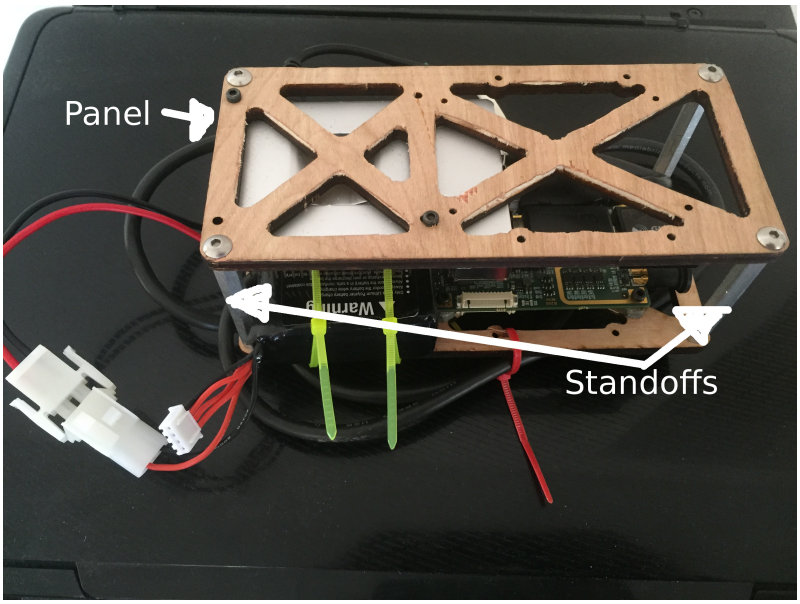


Figure 6.2: Overhead view of the encasing of the MASDR system.

ponents wouldn't split the wood. This enclosure was mounted to the bottom of the drone using the hardware mount on the drone detailed in Section 4.2.

To power the system, an 11V LiPo battery was used in combination with a 5V transformer to step down the voltage for use by the UP Board. The battery and

converter were connected using connectors seen in Figure 6.3, so that the battery could be disconnected when not in use. To prevent any mishandling of plugging the battery in backwards, a male molex connector was soldered to the transformer and a female molex connector was soldered to the battery.

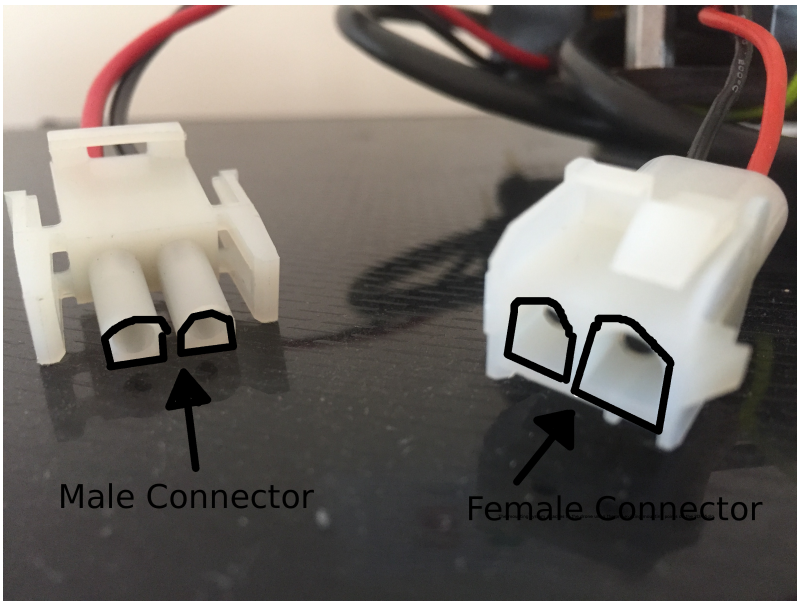


Figure 6.3: Connectors for the battery. The male molex connector (left) is connected to the transformer and the female molex connector (right) is connected to the LiPo battery.

There were two transformers that were purchased, the Nextrox converter was the one that directly converted 12V down to 5V at 3A, and the DROK converter was the other which was an adjustable knob that ranged from 8V-35V to 1.5V-24V at 5A. Unfortunately, the first transformer only worked for the first few trials and the second outright did not work out of the box. The first transformer that was used in the system failed because it became disconnected from the UP Board while power was applied. This transformer was a buck converter which can fail when an output isn't connected due of the failure of a MOSFET or diode [**?**]. The second transformer's adjustable knob did not change the ratio at which the voltage was being output despite multiple attempts and methods. A new transformer was

purchased to replace the malfunctioning transformers.

The UP Board is connected to the 5V transformer using a barrel jack. The male connector was soldered onto the output of the transformer. The UP Board is the main system with everything else connected to it through USB or I2C. It was mounted upside down on the top panel as shown in Figure 6.4.

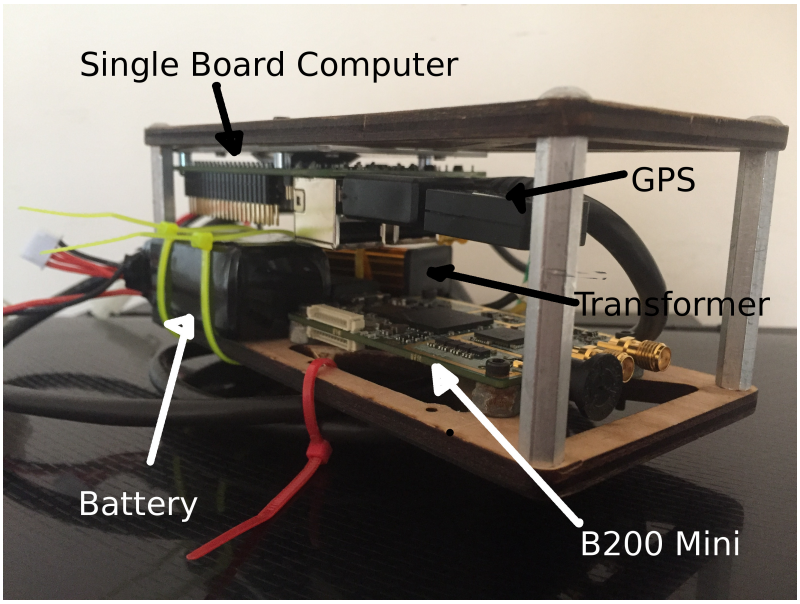


Figure 6.4: Angled side view of the encasing with the UP Board connected to the top panel with the GPS and USB cable to the B200-Mini. The golden transformer and the battery are positioned directly under the UP Board on the bottom panel. The B200-Mini is connected to the bottom panel and is the closest board to the camera.

The GPS is connected to the UP Board using an micro USB to USB adapter. Integration and operation of the GPS proved to be a challenge. The GPSD library that was used to interface with the GPS had permission issues that caused the GPS data to not be read by a C++ program. After the permissions were set correctly, the MASDR program was able to correctly read the GPS data.

The magnetometer was to be connected to the UP Board using the I2C pins. Communication between the UP Board and the magnetometer was never established

after multiple attempts. The UP Board would send a signal to the magnetometer, but never received a response. This could be due to a lack of a Linux driver or a faulty board. A Linux driver for the magnetometer wasn't provided by the manufacturer and the timeline of the project made it unfeasible to write a driver by hand.

The B-200 mini was attached to the bottom panel of the casing. It communicated to the UP Board through a wired USB connection. The 900MHz and 2.4GHz antennas were connected to the SMA connectors on the mini. The 2.4GHz antenna required an adapter because the connector on the board was not compatible with the one on the antenna. During testing, it was realized that the transmission from the board was not working. Upon further inspection it was noticed that a component had broken off the board. Another board was borrowed from a lab to verify that the original board was faulty. The transmission test ran successfully on the board from the lab. A working board and the broken board are shown in the Figure 6.5 and Figure 6.6 respectively.

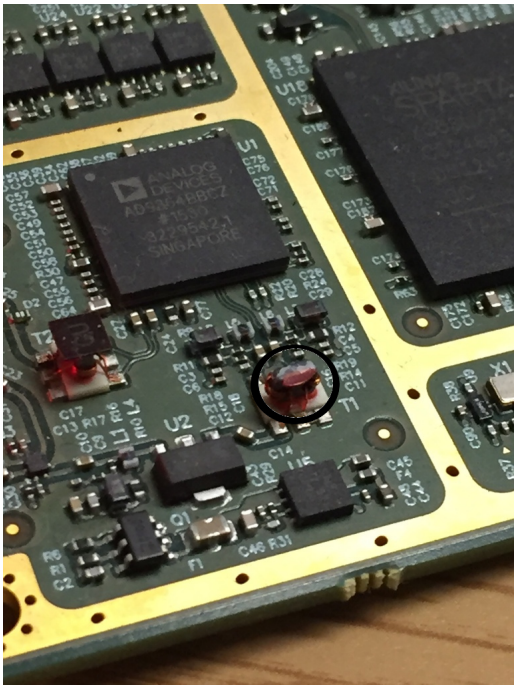


Figure 6.5: The functional B200-mini.

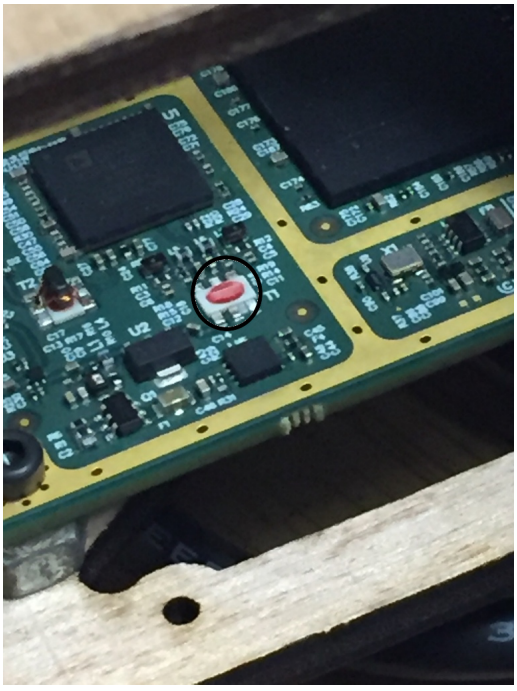


Figure 6.6: The nonfunctional B200-mini.

This component may have broken off during transport to and from meetings or because of vibrations during in flight testing.

The complete system mounted on the drone is shown in Figure 6.7.

Figure 6.7: The complete system mounted on the drone. In flight testing was done with this setup.

6.2 Drone Control

To reduce interference when sensing signals, the WiFi cards on the controller and drone had to be replaced because the communication frequency and the sensing frequency were the same value. In order to replace these WiFi cards, research had to be done to determine which driver was used on the drone. This ended up being the ath9k driver, which was compatible with a wide range of Atheros based cards. Despite the guarantee of compatibility, the first purchase of the R11e-5HnD MikroTik MiniPCI-E wireless card ended up unused, due to it having a large heat sink that could not fit conveniently into both the controller and drone. Furthermore,

the connectors would also require adapters to connect to the antennas. This led to the second purchase of the SparkLAN WPEA-121N MiniPCI-E Half-Size wireless card, which required a bracket to be mounted. The replacement of the WiFi cards in the controller was done first, because the controller acts as a WiFi base station. This makes it easier to test 5 GHz functionality. The replacement went smoothly. However, the controller was still communicating at 2.4 GHz. Upon checking the software, it detected that the WiFi card installed was capable of communicating at 5 GHz. Because of this, the controller's networking scheme was modified such that it would persistently communicate at a specific channel (5GHz). After these changes, there was trouble using SSH to access the controller. One possible explanation is that the antennas are incapable of transmitting at 5 GHz. Limited with time and the scope of the project, it was not possible to obtain antennas to replace on the controller and drone. Due to this reason, the drone's WiFi card was not replaced, and both the controller and drone were factory reset to operate at 2.4GHz.

A Python script was written to briefly take control of the drone and rotate while holding a constant position and altitude. The script used the Dronekit API provided by 3DR. The script was packed up with all necessary libraries and sent to the drone awaiting execution. When the command is given by a computer connected to the controller's WiFi network, the script runs, taking over control of the drone and rotating it. The API did not provide a way to set a constant rate of rotation, so instead a nonblocking call to rotate a certain amount was used in conjunction with timed sleeps to achieve a continuous rotation of 360 degrees. Attempts to map the script to a button on the controller were unsuccessful. The 3DR developers guide notes a future API called S̈mart Shotsẗhat allows commands to be mapped to buttons on the controller, but it was not available at the time of this project.

6.3 Spectrum Sensing

In order to monitor the signal strength of our received data, the RSSI localization technique was used. RSSI was measured by taking the average signal power across a 20 MHz 802.11 wifi channel. This measurement was taken by using the USRP B200 Minis get_rx_sensor command. This command returns the corresponding RSSI value as a double in dBm format. Since a WiFi transmitter is not transmitting all of the time, there are sharp changes in RSSI measurements, as the USRP is receiving both when there is a signal and when there is none.

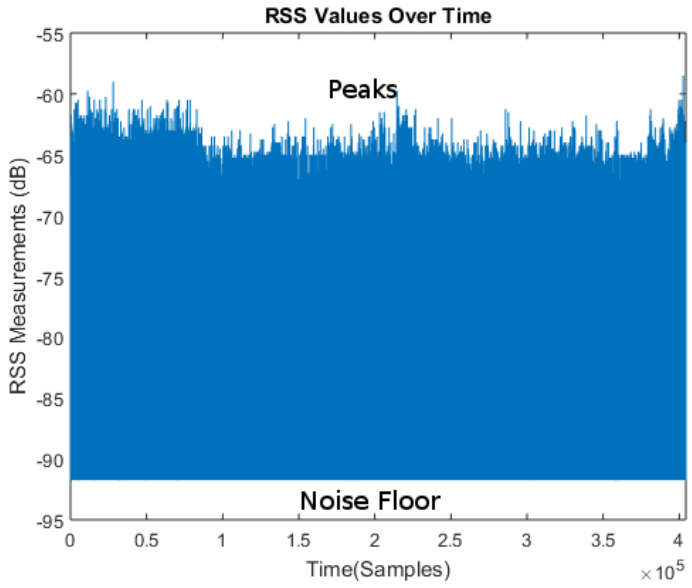


Figure 6.8: This graph shows how the RSSI of our received signal can dramatically change. These sharp transitions in RSSI values are due to the transmitter not always continually transmitting.

In order to have an accurate measurement, RSSI reports that are under the noise floor must be filtered out. A detection threshold of -85 dBm was selected, as it is above the noise floor. The output RSSI measurements only record noticeable signals.

Corresponding RSSI values were used to estimate the distance of the transmission

that was sensed. The localization algorithm depended upon this calculation in order to accurately locate a signal source. the distance equation used was:

$$RSSI = 10\alpha log(d) \tag{6.1}$$

$$d = 10^{(RSSI - RSSI_{calibration})(-10\alpha)} + d_{calibration} \tag{6.2}$$

These equations were mentioned previously in Section 2.3. Based on some field testing, this equation has been fairly accurate.

Table 6.1: The RSSI to distance calculation reasonably relates received signal strength to how far away a transmitter is located.

RSSI (dBm)	Observed Distance (Meters)	Theoretical Distance (Meters)
-50	3	3.2
-60	9	10
-70	26	31.62

When the drone was tested, received signal strengths were between -65 to -80 dBm. These values fall within the expected power range of desired signals, as the sensing platform was 100 ft or 30 meters away from the transmitter.

In addition to calculating RSSI values, a C++ program separate from the code framework was written. This program, called iq_to_file, was created as a foundation with which to work with. It became the testbed for specific elements of MASDR, since it was already capable of logging. The code for this section is in Appendix B.1. This program is built on the rx_samples_to_file example that Ettus Research provides with the UHD. This program already had the desired base functionality, so it proved to be a good starting point. The program was then stripped of unnecessary functionality, including the command-line interface.

With the extraneous functionality removed, components of the MASDR framework were added, in order to test them separately. The matched filter, RSS measurements, and GPS modules were the sections added. This allowed for post-process localization. In order to save these values properly, they were packed into the complex type that was being used to save IQ samples. In order to make it possible to pull these values out in post-processing, they were given an unrealistic imaginary value. with each different module getting a different imaginary value. The matched filter outputs got a flag of 1000. GPS X, Y and Z coordinates were given flags of 2000, 3000, and 4000, respectively. RSS values got a flag of 5000. These values were written with each buffer of samples received. This makes it easier to align results when processing.

Once a data file was recorded, it was processed using a MATLAB script. This script is in Appendix **??**. This script reads in the .dat file produced by iq_to_file, as floats. It then separates the data into in-phase and quadrature components. Then, after pre-allocating buffers, the script pulls out the non-IQ samples. The matched filter values and the RSS measurements are then plotted. The script used to plot the received signal, but with longer record times, this becomes impractical or impossible.

6.4 Spectrum Localization

The GPS was to be used to gather location data to allow for post processing. Unfortunately the GPS was non-functional during the test due to it being unable to locate satellites during the tests. Therefore, in order to use the data gathered in the tests conducted on December 11 and 16, 2016, GPS data was pulled from the drone. This was done using a command-line interface that 3DR provided online [**?**]. Unlike the other logs that were pulled using this method, the GPS information was

logged in a Dataflash log. This format saves the information in MAVlink packets, making it easier to transmit to a base station but harder to post-process. To deal with this, the Ardupilot Mission Planner software was installed [**?**]. This software is capable of taking the dataflash logs generated by the 3DR Solo and converting them into a format that's easier to process. Using this software, the logs for the flights on December 11 and 16, 2016 were converted to MATLAB data files. The relevant GPS information was then pulled from the larger data set.

As a result of the complications with the GPS measurements, the 4-state Kalman filter was modeled in MATLAB. The script can be found in **??**. The script initializes the matrices as described in Section 5.2. A noisy input with a constant x velocity 1 and y velocity 1 is initialized. Arbitrary variance values are used in $\mathbf{P}$ and for the noise covariance matrices $\mathbf{Q}$ and $\mathbf{R}$. Plots of the resulting predictions are shown in Figure 6.11 and Figure 6.12.

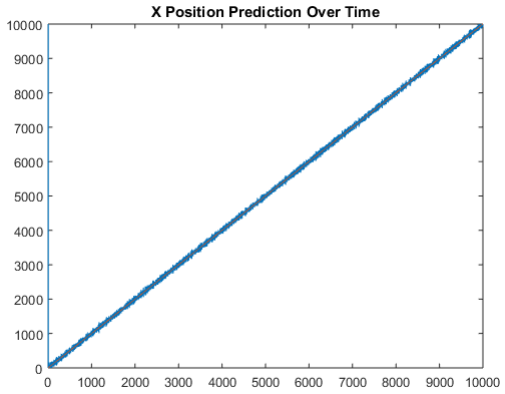


Figure 6.9: X position predictions from Kalman filter simulation.

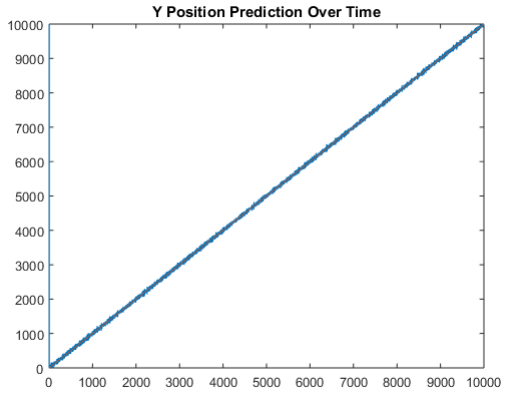


Figure 6.10: Y position predictions from Kalman filter simulation.

Since the test scenario of the Kalman Filter is a constant-velocity model, the expected positions should increase in a linear manner. The resulting predictions that come from the Kalman Filter are decent, but not too much of an improvement. This is due to a number of factors. The first factor is the fact that there are no

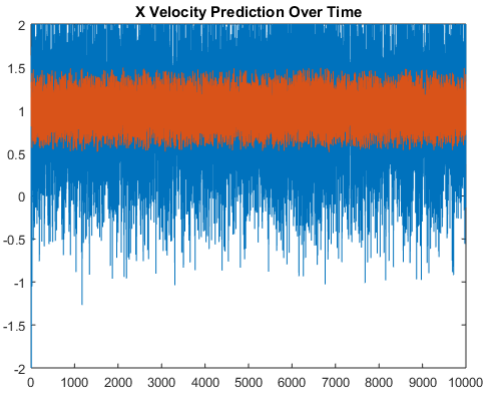


Figure 6.11: X velocity predictions from Kalman filter simulation.

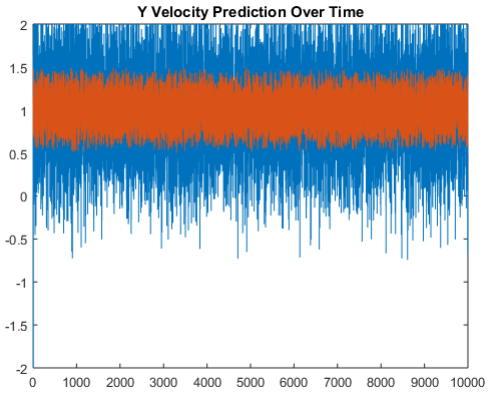


Figure 6.12: Y velocity predictions from Kalman filter simulation.

direct measurements of velocity. Because of this, the resulting velocity measurements already introduce some error. Another factor is the choosing of the various parameters. This was done in a fairly arbitrary manner, due to time constraints.

The script used to display the measured signal strengths was designed for post-proccesing use, taking in a text file of locations and their corresponding RSS values. The processing portion of the script is written in python, and can be found in Appendix **??**. Equation (6.2) is used to calculate the raw distance to the signal. The Pythagorean Theorem is then used to eliminate the altitude component of the distance as can be seen in Figure 6.13. The latitude, longitude, and land-based distance are then formatted into a template string for each point at which a signal was detected.

The formatted strings are inserted into a template html file, primarily composed of a Javascript block that calls into the Google Maps API. Using the drawing tools in the API, rings are drawn on the map corresponding to the calculated distance. An example output screenshot of a generated map has been included in Figure 6.14. The actual output is a webpage with a Google Maps instance running in it, so the map is fully interactive, with the ability to zoom in and scroll around as well.

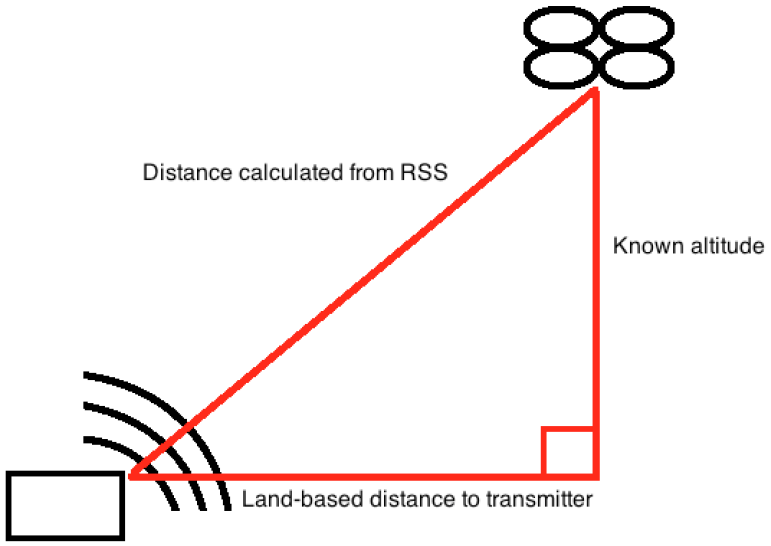


Figure 6.13: Diagram showing right triangle used to calculate land-based distance from the calculated distance and a known altitude.

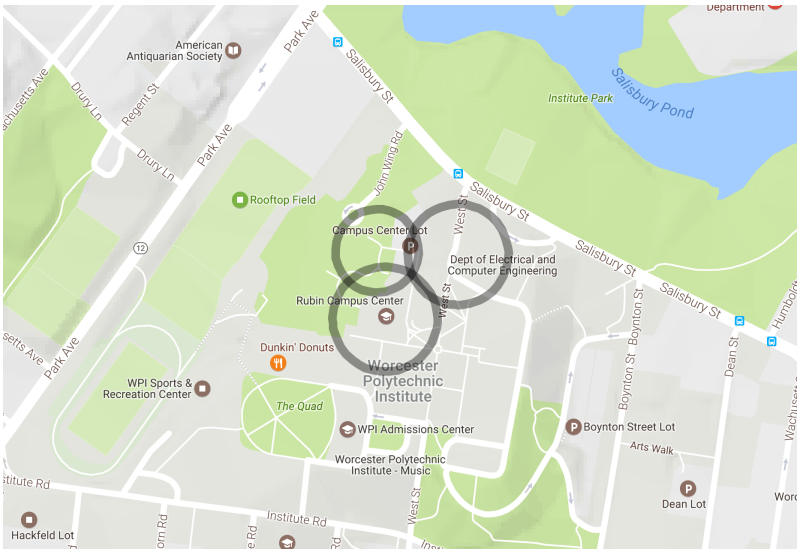


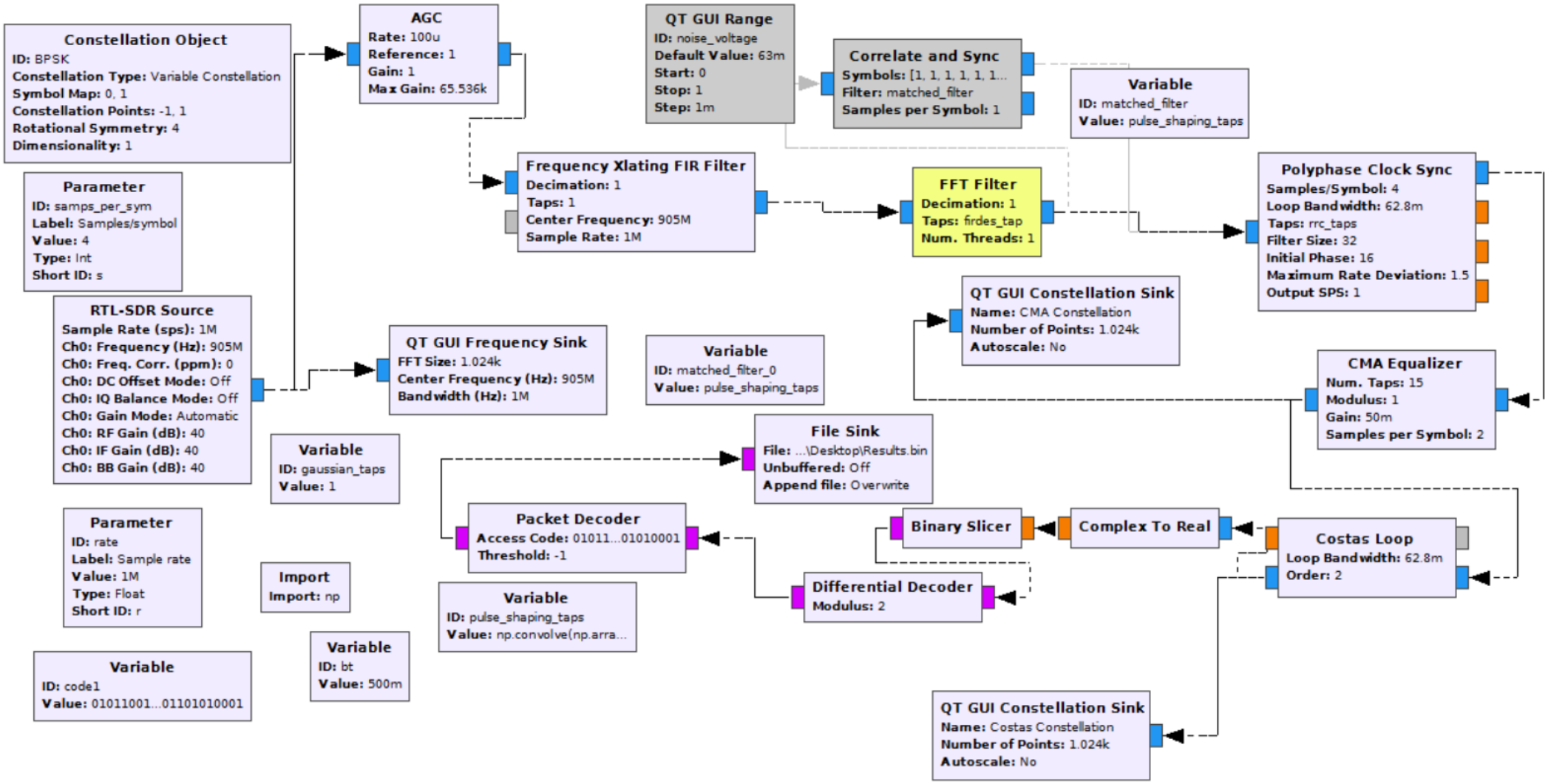
Figure 6.14: Map-based localization example. The rings drawn are calculated from RSS values based on samples taken at the centers of each ring.

6.5 Transmit To Ground

In order to reduce the amount of processing done onboard the drone, a ground receiver was created. It receives important information such as the location, power

level, and frequency in which a signal was detected at. This information is transmitted from the MASDR platform to a ground-based RTL-SDR.

Differential binary phase shift keying or DBPSK modulation was used as the communication protocol for transmission to the ground receiver. This ensures a reduced bit error rate. A DBPSK receiver and transmitter was implemented using GNURadio. The receiver side of the system used the flowgraph shown on the next page. This flowgraph was created in GNU Radio to receive and demodulate the received DBPSK waveform. This is done by using the QT GUI development environment in GNU Radio.


Constellation Object
ID: BPSK
Constellation Type: Variable Constellation
Symbol Map: 0, 1
Constellation Points: -1, 1
Rotational Symmetry: 4
Dimensionality: 1
AGC
Rate: 100u
Reference: 1
Gain: 1
Max Gain: 65.536k
QT GUI Range
ID: noise_voltage
Default Value: 63m
Start: 0
Stop: 1
Step: 1m
Correlate and Sync
Symbols: [1, 1, 1, 1, 1, 1...
Filter: matched_filter
Samples per Symbol: 1
Variable
ID: matched_filter
Value: pulse_shaping_taps
Parameter
ID: samps_per_sym
Label: Samples/symbol
Value: 4
Type: Int
Short ID: s
Frequency Xlating FIR Filter
Decimation: 1
Taps: 1
Center Frequency: 905M
Sample Rate: 1M
FFT Filter
Decimation: 1
Taps: firdes_tap
Num. Threads: 1
Polyphase Clock Sync
Samples/Symbol: 4
Loop Bandwidth: 62.8m
Taps: rrc_taps
Filter Size: 32
Initial Phase: 16
Maximum Rate Deviation: 1.5
Output SPS: 1
QT GUI Constellation Sink
Name: CMA Constellation
Number of Points: 1.024k
Autoscale: No
RTL-SDR Source
Sample Rate (sps): 1M
Ch0: Frequency (Hz): 905M
Ch0: Freq. Corr. (ppm): 0
Ch0: DC Offset Mode: Off
Ch0: IQ Balance Mode: Off
Ch0: Gain Mode: Automatic
Ch0: RF Gain (dB): 40
Ch0: IF Gain (dB): 40
Ch0: BB Gain (dB): 40
QT GUI Frequency Sink
FFT Size: 1.024k
Center Frequency (Hz): 905M
Bandwidth (Hz): 1M
Variable
ID: matched_filter_0
Value: pulse_shaping_taps
CMA Equalizer
Num. Taps: 15
Modulus: 1
Gain: 50m
Samples per Symbol: 2
Variable
ID: gaussian_taps
Value: 1
File Sink
File: ...\Desktop\Results.bin
Unbuffered: Off
Append file: Overwrite
Parameter
ID: rate
Label: Sample rate
Value: 1M
Type: Float
Short ID: r
Packet Decoder
Access Code: 01011...01010001
Threshold: -1
Binary Slicer
Complex To Real
Costas Loop
Loop Bandwidth: 62.8m
Order: 2
Import
Import: np
Differential Decoder
Modulus: 2
Variable
ID: pulse_shaping_taps
Value: np.convolve(np.arra...
Variable
ID: bt
Value: 500m
Variable
ID: code1
Value: 01011001...01101010001
QT GUI Constellation Sink
Name: Costas Constellation
Number of Points: 1.024k
Autoscale: No

This program receives a binary file source which is an array of the data to be sent. This passes through a packet encoder, which encapsulates all of the data from the binary file into a packet with a preamble and an access code, ensuring that the receiver will sync correctly to decode the transmitted data. After this, it goes through a constellation modulator that modulates the binary information into a corresponding analog waveform. It is then transmitted using the USRP with a central frequency of 905 MHz. Before using the receiver, a spectrum monitoring tool for the RTL-SDR called SDR# is used to confirm that the USRP is transmitting on this frequency.

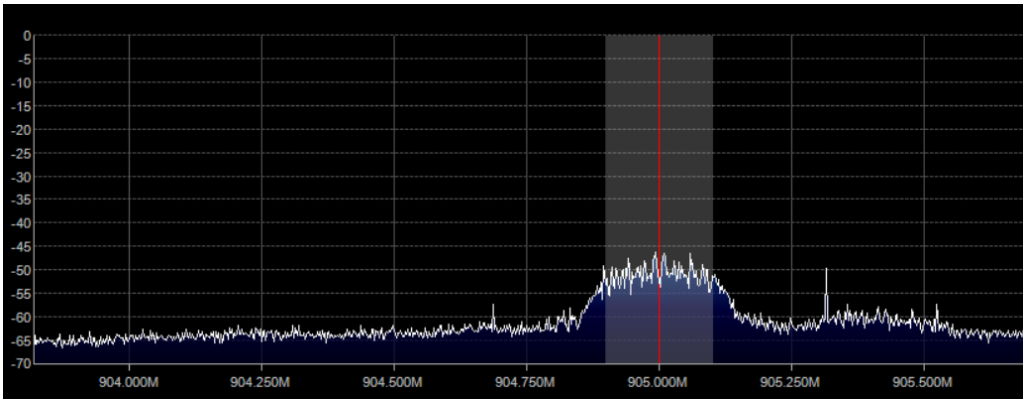


Figure 6.15: An FFT with a center frequency of 905 MHz. This waveform was created by using SDR# on an RTL-SDR.

This signal is then received by the RTL-SDR. However, significant handling of the signal must be taken care of before the correct information can be demodulated. A Frequency Translating FIR Filter was used as a bandpass filter, removing extra signals that might be around that band. After this, a series of three more functions were used. First, a polyphase clock sync is used to perform timing synchronization with the transmitter. This is done by using two filter banks that use a matched filter on a signals pulse shape. A root raised cosine filter is used for our pulse shape since our transmitter is using a root raised cosine filter. The synchronized signal is then

sent through a CMA Equalizer which equalizes modulated baseband signals. Then, a Costas Loop is used in order to provide adequate phase correction so that the received signal's constellation is locked in phase for the entirety of the transmission. The results after all of this signal processing are as expected with the received BPSK signal.

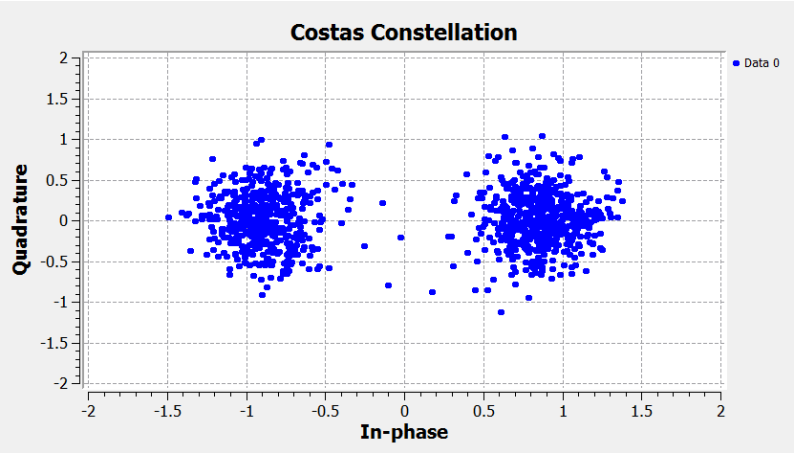


Figure 6.16: A received constellation when the reciever is finished performing the timing offset and phase correction on the transmitted signal. This constellation is taken when the signal strength was at -55 dBm which is the expected signal strength when the drone is receiving signals in the air.

This constellation is sent through a differential decoder and a packet decoder, to extract the payload out of the packetized binary information. This data is then uploaded to results file on the ground receivers main computer.

6.6 Platform Deployment

The platform on the 3DR Solo was tested in the two most applicable scenarios, the rural environment and the urban environment. The tests were conducted in secluded areas to prioritize privacy and safety concerns. The tests consisted of mounting the platform onto the 3DR Solo, flying the drone in a procedural flight path as seen in Figure 6.17 to collect IQ data, and processing the data for the mapping and

the localization of signals around that area. The signals in our case, would be the controller access point.

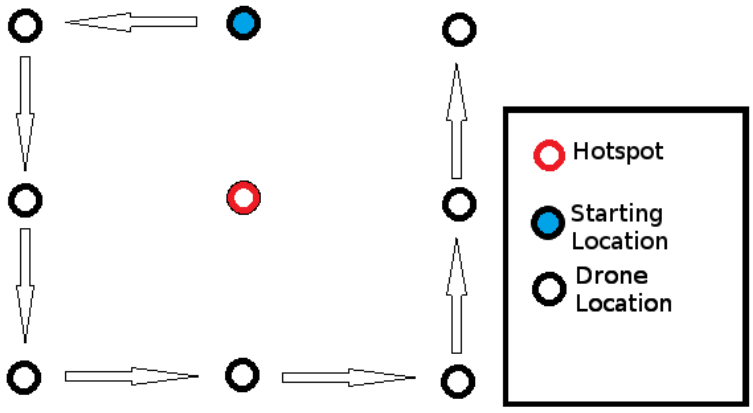


Figure 6.17: 3x3 diagram of test flight plan. The red circle represents where the controller was placed, and the blue circle represents where the drone was started.

The simulation of the rural environment was organized at Professor Wyglinski's property on December 11th, 2016. In order to get the least interfering wifi signals, he turned off his WiFi, and he authorized the flight of the drone on his property. The testing environment consisted of high rise trees that surrounded the area and grew up to an average of 80ft. The resulting mapping and localization of the area is provided in Figure 6.18.

The simulation of the urban environment was organized at the Gateway Park garage on campus, on December 16th, 2016, after having gotten permission for the test from campus police. This WiFi of this area was uncontrolled, with various public and private access points active. This better simulated an urban environment. There were not many tall buildings around, limiting the applicability of the test on denser urban areas. As can be seen in Figure 6.19, the flight kept the platform relatively close to the controller, meaning that the strongest signal in the area would always be the controller. Even so, some of the points further from the controller read higher powers than would be expected from solely the controller, so it's likely that

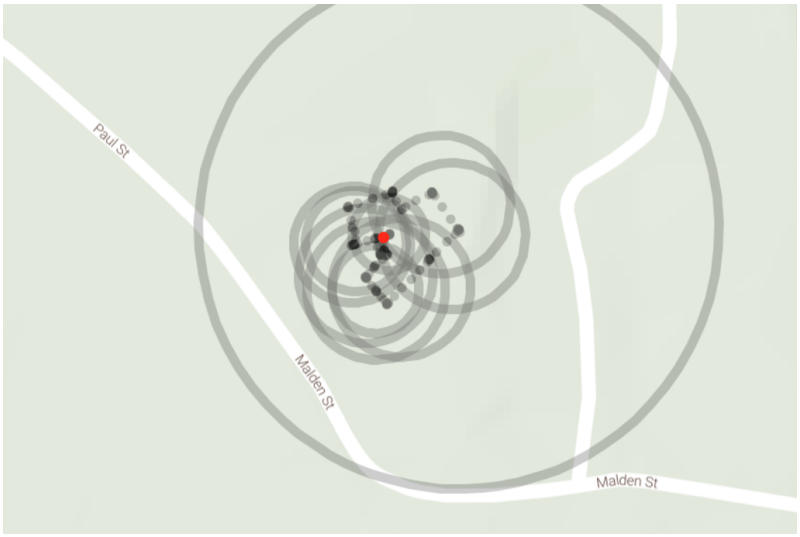


Figure 6.18: The result of the test flight in a rural environment. The red dot shows where the actual access point was located. The black dots represent the GPS locations of where the drone traveled, and the their shades illustrate the length of time the drone was at that location. The black hollow circle depicts the localization of where the drone thinks the access point may be based on GPS and RSSI measurements. The larger hollow circle represents a measurement close to the noise floor.

external Wi-Fi signals influenced the receptions, as was expected. This means that to effectively locate a transmitter in an urban environment, the platform must be closer to that transmitter than any other.

6.7 Chapter Summary

In this chapter, the experimental results achieved in this MQP were discussed. This focused primarily on the design and implementation of the system, as well as the processing of the data collected. Overall, the team was able to create a good foundation on which aerial sensing methods can be applied.

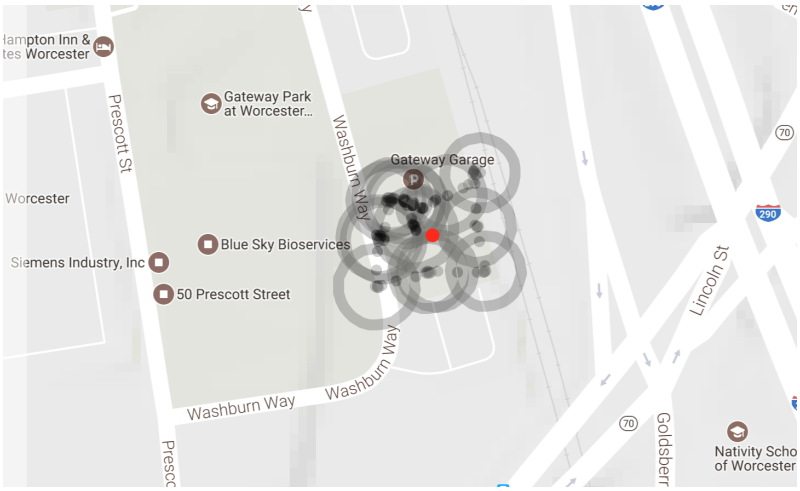


Figure 6.19: The result of the test flight in an urban environment. The red dot shows where the actual access point was located. The black dots represent the GPS locations of where the drone traveled, and the their shades illustrate the length of time the drone was at that location. The black hollow circle depicts the localization of where the drone thinks the access point may be based on GPS and RSSI measurements.

Chapter 7

Conclusions

In this project, drone and SDR technologies were combined, resulting in a versatile platform. By doing so, the team achieved the following:

- **Constructed hardware platform:** Constructed a self-contained SDR signal processing unit, able to operate separate from the drone used.
- **Created software platform:** Created the MASDR software design platform, written to be flexible and easy to modify.
- **Collected data:** Gathered assorted information from the air, including IQ samples, matched filter-processed samples, RSS values, and GPS locations.
- **Communication to ground tested:** Created and tested a DBPSK transmitter and receiver, including Root Raised Cosine Filter and channel equalization.
- **Kalman Filter modeled in MATLAB:** Modelled a four-state tracking Kalman Filter in MATLAB, using a constant-velocity model.

While designing MASDR, a number of problems were encountered. The team came to the following conclusions:

- **Transmission and Receiving should both be done with either UHD or GNURadio.** GNURadio provides a lot of useful blocks that make designing a receiver easier than in UHD, where every block has to be written. However, this means that a lot of the behind-the-scenes work in GNURadio isn't obvious, and has to be deeply investigated. Either use the premade blocks in GNURadio or design the whole system with UHD.

- **GPS is only good for coarse localization.** GPS has a low refresh rate in comparison to the sample rate of the USRP B200 mini. In addition, it only has an precision of a few meters. In order to get a more precise location and estimate, other sensors, like inertial measurement units (IMU), need to be used. This introduces a whole new layer of data fusion.

- **A smarter WiFi Channel decision is needed.** In the MASDR implementation, the SDR only looks at one WiFi channel, in order to allow for accurate RSS measurements. This channel is hard coded into the system. In order to have a more encompassing WiFi sensing and localizing solution, a smarter way to sense WiFi channels is necessary.

7.1 Future Work

The primary focus of this MQP was to create an aerial SDR testbed, on which more specialized projects can be completed. This is the primary area where future work could be based. One possibility is to investigate the localization of the controller of a drone. This involves locating a communication signal from a drone, determining its SSID, finding another transmitter with the same SSID, and localizing it. Another possibility is to use two aerial SDR platforms to empirically model a channel. One platform would act as the transmitter, the other as the receiver. These are two

of many possibilities. With the growth of Cognitive Radio and quadcopters, the number of possible applications will only increase.

One of the secondary purposes of this MQP was to implement WiFi localization on the platform. There are many different ways to do this, covered in the background. This project focused primarily on using a matched filter in the frequency domain to match an OFDM header used by most WiFi communications, but WiFi is only one of many signals to be identified. In looking at other signals, other energy detection techniques may be warranted. In addition, the platform can be modified to classify the signals it receives.

The Kalman filter used in this project was designed for GPS values, and had a few shortcomings. The first shortcoming was that it was only implemented in MATLAB, using a very basic test signal and simplistic model. There is C++ code that was written, but never tested. The first step would be to use the Kalman filter in the existing code. Beyond this, the model used to generate the Kalman gains can be improved, which would result in better predictions. In addition, a Kalman filter could be applied to other aspects of the system, such as the RSS sensing value, and the distance calculation that uses it. This is non-trivial, because the equation for getting distance from RSS is non-linear, so an Extended Kalman filter would be necessary.

Appendix A

Design Configurations

A.1 SBC Selection

Table A.1: SBC Selection

	ODROID-XU4	UP Board	Inforce 6540	EPIA-P90	Jetway NP93	Jetson TK-1
Size	82 x 58 mm	86 x 57 mm		100 x 72 mm	100 x 72 mm	
Weight	38 grams	88 grams				
Power Usage	1-4A	3A	3A			
Power Voltage	5V, 4A	5V, 3A	12V, 3A	12V		
CPU	Exynos5422	Intel x5-Z8350	Snapdragon 805	VIA Eden	Intel N2930	NVIDIA Quad Core
RAM	2GB	4GB	2GB	Up to 8GB	2GB	2GB x 16
USB 3.0	Yes	Yes	Yes	Yes	Yes	Yes
OS	Linux	Windows/Linux		Windows/Linux	Windows/Linux	
Price	$74	$130	$240	$359	$200	$192

A.2 Approach One

Table A.2: Approach One Design Configuration

Hardware Design	**Description**
Drone	3DR Solo
SDR	B-200 Mini
Antenna	Alfa APA-M25
Flight Time	25 min
Flight Speed	5-10 mph
SBC (Computer)	Up Board
Compute Battery	12V 800mAh stepped down to 5V
Payload	700 grams
Storage	32 GB flash drive
Antenna Configuration	Antenna pointed down
Communications Design	**Description**
Drone operation	Human Controlled
Environment to Drone Frequency	2.4 GHz
Controller to Drone TX Frequency	5.7 GHz
Controller to Drone TX Period	Instantaneous (Drone dependent)
Controller to Drone TX Content	Control
Drone to Base TX Frequency	900 MHz
Drone to Base TX Period	2 Hz (Every time drone stops)
Drone to Base TX Content	Location and power level if detected
Signal Processing Design	**Description**
Onboard Computer Processing	Sensing
Offboard Computer Processing	Localization
SDR FPGA Processing	Available if needed
Real Time Localization	No
Sampling Technique	User controlled, record bursts of data
Sampling Space Distance	User Controlled
Number of Samples per Location	Bursts every half second
Localization Method	RSS Triangulation (Average RSS over time)
Energy Sensing Technique	Filter and energy detection

A.3 Approach Two

Table A.3: Approach Two Design Configuration

Hardware Design	**Description**
Drone	DJI S1000
SDR	B-200 Mini
Antenna	Alfa APA-M25
Flight Time	25 min
Flight Speed	5-10 mph
SBC (Computer)	Up Board
Compute Battery	12V 800mAh stepped down to 5V
Payload	2000 grams
Storage	32 GB flash drive
Antenna Configuration	Antenna pointed down
Communications Design	**Description**
Drone operation	Human Controlled
Environment to Drone Frequency	2.4 GHz
Controller to Drone TX Frequency	900 MHz
Controller to Drone TX Period	Instantaneous (Drone dependent)
Controller to Drone TX Content	Control
Drone to Base TX Frequency	5.7 GHz
Drone to Base TX Period	2 Hz (Every time drone stops)
Drone to Base TX Content	Location and power level if detected
Signal Processing Design	**Description**
Onboard Computer Processing	Sensing
Offboard Computer Processing	Localization
SDR FPGA Processing	Available if needed
Real Time Localization	No
Sampling Technique	User controlled, record bursts of data
Sampling Space Distance	User Controlled
Number of Samples per Location	Bursts every half second
Localization Method	RSS Triangulation (Average RSS over time)
Energy Sensing Technique	Filter and energy detection

A.4 Approach Three

Table A.4: Approach Three Design Configuration

Hardware Design	**Description**
Drone	3DR Solo
SDR	B-200 Mini
Antenna	Alfa APA-M25
Flight Time	25 min
Flight Speed	5-10 mph
SBC (Computer)	Up Board
Compute Battery	12V 800mAh stepped down to 5V
Payload	700 grams
Storage	32 GB flash drive
Antenna Configuration	Antenna pointed down
Communications Design	**Description**
Drone operation	Human Controlled
Environment to Drone Frequency	2.4 GHz
Controller to Drone TX Frequency	N/A
Controller to Drone TX Period	N/A
Controller to Drone TX Content	Kill/Return home signal
Drone to Base TX Frequency	900 MHz
Drone to Base TX Period	2 Hz (Every time drone stops)
Drone to Base TX Content	Location and power level if detected
Signal Processing Design	**Description**
Onboard Computer Processing	Sensing, localization, drone control
Offboard Computer Processing	None
SDR FPGA Processing	Available if needed
Real Time Localization	Yes
Sampling Technique	Sample at one spot, move to next spot
Sampling Space Distance	User Controlled
Number of Samples per Location	Bursts every half second
Localization Method	RSS Triangulation (Average RSS over time)
Energy Sensing Technique	Filter and energy detection

A.5 Approach Four

Table A.5: Approach Four Design Configuration

Hardware Design	**Description**
Drone	3DR Solo
SDR	B-200 Mini
Antenna	Alfa APA-M25
Flight Time	25 min
Flight Speed	5-10 mph
SBC (Computer)	Up Board
Compute Battery	12V 800mAh stepped down to 5V
Payload	700 grams
Storage	32 GB flash drive
Antenna Configuration	Sheilding around sensing antenna
Communications Design	**Description**
Drone operation	Human Controlled
Environment to Drone Frequency	2.4 GHz
Controller to Drone TX Frequency	2.4 GHz
Controller to Drone TX Period	Instantaneous (Drone dependent)
Controller to Drone TX Content	Control
Drone to Base TX Frequency	900 MHz
Drone to Base TX Period	2 Hz (Every time drone stops)
Drone to Base TX Content	Location and power level if detected
Signal Processing Design	**Description**
Onboard Computer Processing	Sensing
Offboard Computer Processing	Localization
SDR FPGA Processing	Available if needed
Real Time Localization	No
Sampling Technique	User controlled, record bursts of data
Sampling Space Distance	User Controlled
Number of Samples per Location	Bursts every half second
Localization Method	RSS Triangulation (Average RSS over time)
Energy Sensing Technique	Filter and energy detection

Appendix B

C++ Code

B.1 iq_to_file Source Code

```
//
// Copyright 2010-2011,2014 Ettus Research LLC
// Modified, 2016 by Narut Akadejdechapanich, Scott Iwanicki, Max Li, Kyle Piette,
// Jonas Rogers
//
// This program is free software: you can redistribute it and/or modify
// it under the terms of the GNU General Public License as published by
// the Free Software Foundation, either version 3 of the License, or
// (at your option) any later version.
//
// This program is distributed in the hope that it will be useful,
// but WITHOUT ANY WARRANTY; without even the implied warranty of
// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
// GNU General Public License for more details.
//
// You should have received a copy of the GNU General Public License
// along with this program.  If not, see <http://www.gnu.org/licenses/>.
//

#include <uhd/types/tune_request.hpp>
#include <uhd/utils/thread_priority.hpp>
#include <uhd/utils/safe_main.hpp>
#include <uhd/usrp/multi_usrp.hpp>
#include <uhd/exception.hpp>
#include <boost/program_options.hpp>
#include <boost/format.hpp>
#include <boost/thread.hpp>
#include <iostream>
#include <fstream>
#include <csignal>
```

```
#include <complex>
#include <fftw3.h>
#include <pthread.h>
#include <gps.h>
#include <unistd.h>
#include <math.h>

#define N_FFT 1024
#define GPS_BUF_SIZE 60  // Hold the past 6 seconds of samples

namespace po = boost::program_options;

///FFT handles, arrays
fftw_complex *fft_in, *fft_out; ///< Buffers for FFT.
fftw_plan fft_p;
fftw_complex ofdm_head[N_FFT]; //< Expected OFDM Header.

///Interrupt Handlers
static bool stop_signal_called = false;
void sig_int_handler(int){stop_signal_called = true;}



//GPS values
pthread_t gps_thread;
int gps_running = 1; // flag to start/stop gps polling
                     // need to recall poll_gps after setting flag back to 1
double gps_buff[GPS_BUF_SIZE][3]; //GPS data buffer.
                                  //0 is latitude
                                  //1 is longitude
                                  //2 is timestamp
volatile int gps_buf_head = 0; //current gps buffer head
struct gps_data_t gps_data__;    //GPS struct
/*************************************************************************/
/*************************GPS FUNCTIONS***********************************/
/*************************************************************************/
int init_gps(){
    int rc,x;
    gps_running =1;
    if ((rc = gps_open("localhost", "2947", &gps_data__)) == -1) {
        printf("code: %d, reason: %s\n", rc, gps_errstr(rc));
        gps_running = 0;
        return 0;
    }

    //set gps stream to watch JSON
    gps_stream(&gps_data__, WATCH_ENABLE | WATCH_JSON, NULL);

    //initialize gps buffer to 0s
    for(rc=0; rc<GPS_BUF_SIZE; rc++){
        for(x=0; x<3; x++){
            gps_buff[rc][x] = 0;
        }
```

```
84      }
85      return 1;
86  }
87
88
89  /****************************************************************************/
90  void *poll_gps(void *unused){
91      int rc;
92      while (gps_running) {
93          // wait for 2 seconds to receive data
94          if (gps_waiting (&gps_data__, 2000000)) {
95              /* read data */
96              if ((rc = gps_read(&gps_data__)) == -1) {
97                  printf("error occured reading gps data. code: %d, reason: %s\n",
98                         rc, gps_errstr(rc));
99              } else {
100
101                 // Write data from the GPS receiver
102                 if ((gps_data__.status == STATUS_FIX) &&
103                   (gps_data__.fix.mode == MODE_2D || gps_data__.fix.mode == MODE_3D) &&
104                   !isnan(gps_data__.fix.latitude) && !isnan(gps_data__.fix.longitude)) {
105
106                     gps_buff[gps_buf_head][0] = gps_data__.fix.latitude;
107                     gps_buff[gps_buf_head][1] = gps_data__.fix.longitude;
108                     gps_buff[gps_buf_head][2] = gps_data__.fix.time;
109                     //Loop buffer
110                     gps_buf_head = (gps_buf_head + 1) % GPS_BUF_SIZE;
111
112                 } else {
113                     printf("no GPS data available\n");
114                 }
115             }
116         }
117     }
118     pthread_exit(NULL);
119 }
120
121 /****************************************************************************/
122
123 void get_gps_data(double *latitude, double *longitude, double *time){
124     int pos = (gps_buf_head - 1) % GPS_BUF_SIZE;
125     *latitude  = gps_buff[pos][0];
126     *longitude = gps_buff[pos][1];
127     *time      = gps_buff[pos][2];
128     return;
129 }
130
131
132 /****************************************************************************/
133
134 void rem_gps(){
135     gps_running = 0;
136     gps_stream(&gps_data__, WATCH_DISABLE, NULL);
```

```
137     gps_close (&gps_data__);
138 }
139 
140 /***********************************************************************/
141 /*************************RECV, FS WRITING FUNCTIONS*********************/
142 /***********************************************************************/
143 template<typename samp_type> void recv_to_file(
144     uhd::usrp::multi_usrp::sptr usrp,
145     const std::string &cpu_format,
146     const std::string &wire_format,
147     const std::string &file,
148     size_t samps_per_buff,
149     unsigned long long num_requested_samples,
150     double time_requested = 0.0,
151     bool null = false,
152     bool continue_on_bad_packet = false
153 ){
154     int i;
155     unsigned long long num_total_samps = 0;
156     double lat_in, long_in,time_in;
157     float match_val[2] = {0,0}, re, im;
158     std::complex<samp_type> match_mag, lat_val, long_val, time_val, rssi_val;
159     double rssi = 0;
160     boost::this_thread::sleep(boost::posix_time::seconds(2)); //allow for some setup time
161     //create a receive streamer
162     uhd::stream_args_t stream_args(cpu_format,wire_format);
163     uhd::rx_streamer::sptr rx_stream = usrp->get_rx_stream(stream_args);
164     uhd::rx_metadata_t md;
165 
166     // extra size for match filt value, latitude, longitude, time
167     std::complex<samp_type> buff[samps_per_buff + 5];
168 
169     //File handle
170     std::ofstream outfile;
171     if (not null)
172         outfile.open(file.c_str(), std::ofstream::binary);
173     bool overflow_message = true;
174 
175     //setup streaming
176     uhd::stream_cmd_t stream_cmd((num_requested_samples == 0)?
177         uhd::stream_cmd_t::STREAM_MODE_START_CONTINUOUS:
178         uhd::stream_cmd_t::STREAM_MODE_NUM_SAMPS_AND_DONE
179     );
180     stream_cmd.num_samps = size_t(num_requested_samples);
181     stream_cmd.stream_now = true;
182     stream_cmd.time_spec = uhd::time_spec_t();
183     rx_stream->issue_stream_cmd(stream_cmd);
184 
185     boost::system_time start = boost::get_system_time();
186     unsigned long long ticks_requested = (long)(time_requested * (double)boost::posix_time::
      time_duration::ticks_per_second());
187     boost::posix_time::time_duration ticks_diff;
188     boost::system_time last_update = start;
```

```
189     unsigned long long last_update_samps = 0;
190
191     typedef std::map<size_t,size_t> SizeMap;
192     SizeMap mapSizes;
193     std::cout<< "enter while loop"<<std::endl;
194     while(not stop_signal_called and (num_requested_samples != num_total_samps or
        num_requested_samples == 0)) {
195         boost::system_time now = boost::get_system_time();
196
197         size_t num_rx_samps = rx_stream->recv(buff, samps_per_buff, md, 3.0, 0);
198
199         if (md.error_code == uhd::rx_metadata_t::ERROR_CODE_TIMEOUT) {
200             std::cout << boost::format("Timeout while streaming") << std::endl;
201             break;
202         }
203         if (md.error_code == uhd::rx_metadata_t::ERROR_CODE_OVERFLOW){
204             if (overflow_message) {
205                 overflow_message = false;
206                 std::cerr << boost::format(
207                     "Got an overflow indication. Please consider the following:\n"
208                     "  Your write medium must sustain a rate of %fMB/s.\n"
209                     "  Dropped samples will not be written to the file.\n"
210                     "  Please modify this example for your purposes.\n"
211                     "  This message will not appear again.\n"
212                 ) % (usrp->get_rx_rate()*sizeof(std::complex<samp_type>)/1e6);
213             }
214             continue;
215         }
216         if (md.error_code != uhd::rx_metadata_t::ERROR_CODE_NONE){
217             std::string error = str(boost::format("Receiver error: %s") % md.strerror());
218             if (continue_on_bad_packet){
219                 std::cerr << error << std::endl;
220                 continue;
221             }
222             else
223                 throw std::runtime_error(error);
224         }
225
226         num_total_samps += num_rx_samps;
227
228         //Copy buff.front() into an FFT
229         for(i = 0; i < N_FFT; i++){
230             fft_in[i][0] = buff[i].real();
231             fft_in[i][0] = buff[i].imag();
232         }
233
234         //Execute FFT
235         fftw_execute(fft_p);
236
237         //Matched filter.
238         for(i = 0; i < N_FFT; i++) {
239             re = fft_out[i][0] * ofdm_head[i][0] - fft_out[i][1] * ofdm_head[i][1];
240             im = fft_out[i][0] * ofdm_head[i][1] + fft_out[i][1] * ofdm_head[i][0];
```

```
241             match_val[0] += re;
242             match_val[1] += im;
243         }
244         //Read RSS value.
245         rssi = usrp->get_rx_sensor("rssi",0).to_real();
246 
247         //Use imaginary values in complex data type to flag non-sample values.
248         get_gps_data(&lat_in,&long_in,&time_in);
249         match_mag = std::complex<samp_type>(sqrt(match_val[0]*match_val[0] + \
250                                                  match_val[1]*match_val[1]),1000);
251         lat_val = std::complex<samp_type>((samp_type)lat_in,2000);
252         long_val = std::complex<samp_type>((samp_type)long_in,3000);
253         time_val = std::complex<samp_type>((samp_type)time_in,4000);
254         rssi_val = std::complex<samp_type>((samp_type)rssi,5000);
255 
256         //Add results to write buffer
257         buff[samps_per_buff] = match_mag;
258         buff[samps_per_buff+1] = lat_val;
259         buff[samps_per_buff+2] = long_val;
260         buff[samps_per_buff+3] = time_val;
261         buff[samps_per_buff+4] = rssi_val;
262 
263         //Write to buffer.
264         if (outfile.is_open())
265             outfile.write((const char*)buff, (num_rx_samps+5)*sizeof(std::complex<samp_type>));
266 
267         ticks_diff = now - start;
268         if (ticks_requested > 0){
269             if ((unsigned long long)ticks_diff.ticks() > ticks_requested)
270                 break;
271         }
272     }
273 
274     stream_cmd.stream_mode = uhd::stream_cmd_t::STREAM_MODE_STOP_CONTINUOUS;
275     rx_stream->issue_stream_cmd(stream_cmd);
276 
277     if (outfile.is_open())
278         outfile.close();
279 }
280 
281 /***********************************************************************/
282 typedef boost::function<uhd::sensor_value_t (const std::string&)> get_sensor_fn_t;
283 
284 bool check_locked_sensor(std::vector<std::string> sensor_names, const char* sensor_name,
      get_sensor_fn_t get_sensor_fn, double setup_time){
285     if (std::find(sensor_names.begin(), sensor_names.end(), sensor_name) == sensor_names.end())
286         return false;
287 
288     boost::system_time start = boost::get_system_time();
289     boost::system_time first_lock_time;
290 
291     std::cout << boost::format("Waiting for \"%s\": ") % sensor_name;
292     std::cout.flush();
```

```
293
294     while (true) {
295         if ((not first_lock_time.is_not_a_date_time()) and
296                 (boost::get_system_time() > (first_lock_time + boost::posix_time::seconds(
      setup_time))))
297         {
298             std::cout << " locked." << std::endl;
299             break;
300         }
301         if (get_sensor_fn(sensor_name).to_bool()){
302             if (first_lock_time.is_not_a_date_time())
303                 first_lock_time = boost::get_system_time();
304             std::cout << "+";
305             std::cout.flush();
306         }
307         else {
308             first_lock_time = boost::system_time(); //reset to 'not a date time'
309
310             if (boost::get_system_time() > (start + boost::posix_time::seconds(setup_time))){
311                 std::cout << std::endl;
312                 throw std::runtime_error(str(boost::format("timed out waiting for consecutive
      locks on sensor \"%s\"") % sensor_name));
313             }
314             std::cout << "_";
315             std::cout.flush();
316         }
317         boost::this_thread::sleep(boost::posix_time::milliseconds(100));
318     }
319     std::cout << std::endl;
320     return true;
321 }
322
323 int UHD_SAFE_MAIN(int argc, char *argv[]){
324     uhd::set_thread_priority_safe();
325
326
327
328     //Initialize OFDM_head
329    int i;
330     for(i=0;i<N_FFT;i++){
331         int ratio = N_FFT/64; //Number of fft bins in an OFDM bin.
332
333         //Account for the fact that only middle 52 OFDM bins are used.
334         if (i < 6 * ratio || i > 58 * ratio){
335             ofdm_head[i][0] = 0;
336             ofdm_head[i][1] = 0;
337         }
338         else if(i >= (6+6) * ratio && i < (6+7) * ratio){
339             ofdm_head[i][0] = 1;
340             ofdm_head[i][1] = 1;
341         }
342         else if(i >= (12+14) * ratio && i < (12+15) * ratio) {
343             ofdm_head[i][0] = 1;
```

```
344             ofdm_head[i][1] = 1;
345         }
346         else if(i >= (26+14) * ratio && i < (26+15) * ratio) {
347             ofdm_head[i][0] = 1;
348             ofdm_head[i][1] = 1;
349         }
350         else if(i >= (40+14) * ratio && i < (40 + 15) * ratio){
351             ofdm_head[i][0] = 1;
352             ofdm_head[i][1] = 1;
353         }
354         else {
355             ofdm_head[i][0] = 0;
356             ofdm_head[i][1] = 0;
357         }
358     }
359     //Initialize FFT.
360     fft_in = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * N_FFT);
361     fft_out = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * N_FFT);
362     fft_p = fftw_plan_dft_1d(N_FFT, fft_in, fft_out, FFTW_FORWARD, FFTW_MEASURE);
363
364     //Initialize USRP
365     std::string file = "usrp_samples.dat",  \
366                 type = "float",             \
367                 ant = "RX2",                \
368                 ref = "internal",           \
369                 wirefmt = "sc16",           \
370                 args;
371
372     short       total_num_samps = 0,        \
373                 total_time = 0,             \
374                 spb=N_FFT,                  \
375                 setup_time=1,               \
376                 gain=40;
377
378     double      rate = 1e6,                 \
379                 freq=2.4e9;
380
381     bool    null = 0,                       \
382             continue_on_bad_packet = 0;
383
384
385     //create a usrp device
386     uhd::usrp::multi_usrp::sptr usrp = uhd::usrp::multi_usrp::make(args);
387
388     //Lock mboard clocks
389     usrp->set_clock_source(ref);
390
391     //set the sample rate
392     usrp->set_rx_rate(rate);
393
394     uhd::tune_request_t tune_request(freq);
395     usrp->set_rx_freq(tune_request);
396
```

```
397     usrp->set_rx_gain(gain);
398
399     //set the antenna
400     usrp->set_rx_antenna(ant);
401     boost::this_thread::sleep(boost::posix_time::seconds(setup_time)); //allow for some setup
        time
402
403     //check Ref and LO Lock detect
404
405     check_locked_sensor(usrp->get_rx_sensor_names(0), "lo_locked", boost::bind(&uhd::usrp::
        multi_usrp::get_rx_sensor, usrp, _1, 0), setup_time);
406
407
408     if (total_num_samps == 0){
409         std::signal(SIGINT, &sig_int_handler);
410         std::cout << "Press Ctrl + C to stop streaming..." << std::endl;
411     }
412
413     std::cout<<"Init GPS"<<std::endl;
414
415     int rc;
416     void * temp;
417     init_gps();
418     rc = pthread_create(&gps_thread, NULL, poll_gps, temp);
419
420     if (rc){
421         std::cout << "Error:unable to create thread," << rc << std::endl;
422     }
423     boost::this_thread::sleep(boost::posix_time::seconds(2)); //allow for some setup time
424     std::cout<< "Ready!" <<std::endl;
425 #define recv_to_file_args(format) \
426     (usrp, format, wirefmt, file, spb, total_num_samps, total_time, null, continue_on_bad_packet
        )
427     //recv to file
428     if (type == "double") recv_to_file<double >recv_to_file_args("fc64");
429     else if (type == "float") recv_to_file<float>recv_to_file_args("fc32");
430     else if (type == "short") recv_to_file<short>recv_to_file_args("sc16");
431     else throw std::runtime_error("Unknown type " + type);
432
433     //finished
434     std::cout << std::endl << "Done!" << std::endl << std::endl;
435
436     return EXIT_SUCCESS;
437 }
```

B.2 MASDR Code

B.2.1 utils.h

```
// File: utils.h
//
// MASDR Project 2016
// WPI MQP E-Project number:
// Members: Jonas Rogers
//          Kyle Piette
//          Max Li
//          Narut Akadejdechapanich
//          Scott Iwanicki
// Advisor: Professor Alex Wyglinski
// Sponsor: Gryphon Sensors

#ifndef __utils_h__
#define __utils_h__

// Standard Libraries
#include <iostream>
#include <stdio.h>
#include <errno.h>
#include <string.h>
// UHD Libraries
#include <uhd/usrp/multi_usrp.hpp>
// Boost Libraries
#include <boost/format.hpp>
#include <boost/thread.hpp>
// GPS libraries
#include <gps.h>
#include <unistd.h>
#include <math.h>
#include <pthread.h> //Using pthread for GPS

// SDR buffer sizes
#define RBUF_SIZE 16384
#define TBUF_SIZE 162   //11/6/16 NARUT: dependent
                        //on our packet size 5 floats (32*3) + (33*2) start/end
#define SPS 4 //4 samples per symbol.

//GPS constants
#define GPS_BUF_SIZE 60  // Hold the past 6 seconds of samples

// Energy detection constants
#define THRESH_E 0.1 ///11/14/16 MHLI: Picked based on received information.
#define THRESH_MATCH 0 //11/14/16 MHLI: 20,25 would work probably, especially in

// Standard defines
#define PI       3.14159265359
#define BIT0     0x00000001
#define BIT1     0x00000002
```

```
#define BIT2    0x00000004
#define BIT3    0x00000008
#define BIT4    0x00000010
#define BIT5    0x00000020
#define BIT6    0x00000040
#define BIT7    0x00000080
#define BIT8    0x00000100
#define BIT9    0x00000200
#define BIT10   0x00000400
#define BIT11   0x00000800
#define BIT12   0x00001000
#define BIT13   0x00002000
#define BIT14   0x00004000
#define BIT15   0x00008000
#define BIT16   0x00010000
#define BIT17   0x00020000
#define BIT18   0x00040000
#define BIT19   0x00080000
#define BIT20   0x00100000
#define BIT21   0x00200000
#define BIT22   0x00400000
#define BIT23   0x00800000
#define BIT24   0x01000000
#define BIT25   0x02000000
#define BIT26   0x04000000
#define BIT27   0x08000000
#define BIT28   0x10000000
#define BIT29   0x20000000
#define BIT30   0x40000000
#define BIT31   0x80000000

/**
 * Structure for received sample buffer and heading.
 */
typedef struct samp_block {
    float heading; ///< Heading in degrees from north, according to magnetometer
    std::complex<float> samples[RBUF_SIZE]; ///< USRP samples
} Sampblock;

/**
* Structure for GPS Position
*/
typedef struct gps_dat {
    float x;
    float y;
    float v_x;
    float v_y;
    float e_x;
    float e_y;
} GPSData;

/**
 * Linked list node structure of data to be packaged then transmitted.
```

```
 */
typedef struct transnode {
    float heading; ///< Heading in degrees from north, according to magnetometer
    float gps[3]; ///< The GPS location for this data
    float data; ///< The data that we want to transmit
    struct transnode* next; ///< Next recorded block, either a pointer or NULL

    /**
     * @brief Delete the next item in the list.
     *
     * This will call recursively until everything after the element it was
     * initially called on is deleted.
     */
    ~transnode() {
        delete next;
    }
} TransNode;

/**
 * Status of the software on the SBC.
 */
typedef enum {
    SAMPLE,
    PROCESS,
    TRANSMIT,
    IDLE,
} SoftStatus;

/**
 * Physical status of the platform.
 *
 * Includes location, heading, and stationarity.
 */
typedef struct {
    double location[3]; ///< Location as an array of lat, long, and height.
    double heading; ///< Heading in degrees from north.
    bool is_stat_and_rot; ///< Currently stationary and rotating.
} PhyStatus;

/**
 * Structure for header message before data transmission.
 *
 * Includes sampling location and number of hits, indicating how many TxHit
 * messages will be following.
 */
typedef struct {
    unsigned int tx_id; ///< Transmission ID number to link with data packets.
    double location[3]; ///< Location of this sampling session.
    int num_hits; ///< Number of signals detected. (Need to discuss and clarify)
} TxHeader;

/**
 * Structure for transmission of data concerning a single detected signal.
```

```
 */
typedef struct {
    unsigned int tx_id; ///< Transmission ID number to link with header packet.
    double heading; ///< Heading in degrees from North of detected signal.
    double strength; ///< Strength of detected signal.
} TxHit;

/**
 * @brief Handle a SIGINT nicely.
 */
void handle_sigint(int);

/**
 * @brief Test FFT functionality.
 */
void fft_test();

/**
 * type provided by UHD, find documentation at http://files.ettus.com/manual/
 */
typedef boost::function<uhd::sensor_value_t (const std::string&)>
    get_sensor_fn_t;

/**
 * Function provided by UHD.
 *
 * Documentation at http://files.ettus.com/manual/
 */
bool check_locked_sensor(std::vector<std::string> sensor_names,
                         const char* sensor_name,
                         get_sensor_fn_t get_sensor_fn,
                         double setup_time);

/**
 * Initialize USB GPS
 */
int init_gps();

/**
 * Reads latitude, longitude, and time from GPS and puts it in FIFO
 */
void *poll_gps();

/**
 * Read data from GPS FIFO
 */
void get_gps_data(double *latitude, double *longitude, double *time);

/**
 * Shuts down GPS
 */
void rem_gps();

```

```
#endif // __utils_h__
```

B.2.2 utils.cpp

```
// File: utils.cpp
//
// MASDR Project 2016
// WPI MQP E-Project number:
// Members: Jonas Rogers
//          Kyle Piette
//          Max Li
//          Narut Akadejdechapanich
//          Scott Iwanicki
// Advisor: Professor Alex Wyglinski
// Sponsor: Gryphon Sensors

#include "utils.h"
static bool stop_signal_called=false; ///< Global for keyboard interrupts

int gps_running = 1; // flag to start/stop gps polling
                     // need to recall poll_gps after setting flag back to 1
double gps_buff[GPS_BUF_SIZE][3]; //GPS data buffer.
                                  //0 is latitude
                                  //1 is longitude
                                  //2 is timestamp
volatile int gps_buf_head = 0; //current gps buffer head
struct gps_data_t gps_data__;    //GPS struct


/*****************************************************************************/
bool check_locked_sensor(std::vector<std::string> sensor_names,
                         const char* sensor_name,
                         get_sensor_fn_t get_sensor_fn,
                         double setup_time){
    if (std::find(sensor_names.begin(),
                  sensor_names.end(),
                  sensor_name) == sensor_names.end()) {
        return false;
    }

    boost::system_time start = boost::get_system_time();
    boost::system_time first_lock_time;

    std::cout << boost::format("Waiting for \"%s\": ") % sensor_name;
    std::cout.flush();

    while(1) {
        if (!first_lock_time.is_not_a_date_time()
            && (boost::get_system_time()
                > (first_lock_time + boost::posix_time::seconds(setup_time)))) {
            std::cout << " locked." << std::endl;
            break;
```

```
        }
        if (get_sensor_fn(sensor_name).to_bool()) {
            if (first_lock_time.is_not_a_date_time()) {
                first_lock_time = boost::get_system_time();
            }
            std::cout << "+" << std::flush;
        }
        else {
            first_lock_time = boost::system_time(); //reset to 'not a date time'

            if (boost::get_system_time()
                > (start + boost::posix_time::seconds(setup_time))) {
                std::cout << std::endl;
                throw std::runtime_error(str(boost::format(
                    "Timed out waiting for consecutive locks on sensor \"%s\""
                    ) % sensor_name));
            }
            std::cout << "-" << std::flush;
        }
        boost::this_thread::sleep(boost::posix_time::milliseconds(100));
    }
    std::cout << std::endl;
    return true;
}

/***********************************************************************/
void handle_sigint(int) {
    stop_signal_called = true;
    rem_gps();
    exit(0);
}

/***********************************************************************/
int init_gps(){
    int rc,x;
    gps_running =1;
    if ((rc = gps_open("localhost", "2947", &gps_data__)) == -1) {
        printf("code: %d, reason: %s\n", rc, gps_errstr(rc));
        gps_running = 0;
        return 0;
    }

    //set gps stream to watch JSON
    gps_stream(&gps_data__, WATCH_ENABLE | WATCH_JSON, NULL);

    //initialize gps buffer to 0s
    for(rc=0; rc<GPS_BUF_SIZE; rc++){
        for(x=0; x<3; x++){
            gps_buff[rc][x] = 0;
        }
    }
    return 1;
}
```

```
102
103
104 /*****************************************************************************/
105 void *poll_gps(void *unused){
106     int rc;
107     while (gps_running) {
108         // wait for 2 seconds to receive data
109         if (gps_waiting (&gps_data__, 2000000)) {
110             /* read data */
111             if ((rc = gps_read(&gps_data__)) == -1) {
112                 printf("error occured reading gps data. code: %d, reason: %s\n",
113                        rc, gps_errstr(rc));
114             } else {
115
116                 // Write data from the GPS receiver
117                 if ((gps_data__.status == STATUS_FIX) &&
118                   (gps_data__.fix.mode == MODE_2D || gps_data__.fix.mode == MODE_3D) &&
119                   !isnan(gps_data__.fix.latitude) && !isnan(gps_data__.fix.longitude)) {
120
121                     gps_buff[gps_buf_head][0] = gps_data__.fix.latitude;
122                     gps_buff[gps_buf_head][1] = gps_data__.fix.longitude;
123                     gps_buff[gps_buf_head][2] = gps_data__.fix.time;
124
125                     //Loop buffer
126                     gps_buf_head = (gps_buf_head + 1) % GPS_BUF_SIZE;
127
128                 } else {
129                     printf("no GPS data available\n");
130                 }
131             }
132         }
133     }
134     pthread_exit(NULL);
135 }
136
137 /*****************************************************************************/
138
139 void get_gps_data(double *latitude, double *longitude, double *time){
140     int pos = (gps_buf_head - 1) % GPS_BUF_SIZE;
141     *latitude  = gps_buff[pos][0];
142     *longitude = gps_buff[pos][1];
143     *time      = gps_buff[pos][2];
144     return;
145 }
146
147
148 /*****************************************************************************/
149
150 void rem_gps(){
151     gps_running = 0;
152     gps_stream(&gps_data__, WATCH_DISABLE, NULL);
153     gps_close (&gps_data__);
154 }
```

B.2.3 masdr.h

```
// File: masdr.h
//
// MASDR Project 2016
// WPI MQP E-Project number:
// Members: Jonas Rogers
//          Kyle Piette
//          Max Li
//          Narut Akadejdechapanich
//          Scott Iwanicki
// Advisor: Professor Alex Wyglinski
// Sponsor: Gryphon Sensors

#ifndef __masdr_h__
#define __masdr_h__

// Standard Libraries
#include <iostream>
#include <csignal>
#include <complex>
#include <cmath>
// UHD Libraries
#include <uhd/types/tune_request.hpp>
#include <uhd/utils/thread_priority.hpp>
#include <uhd/utils/safe_main.hpp>
#include <uhd/usrp/multi_usrp.hpp>
#include <uhd/exception.hpp>
// FFT Library
#include <fftw3.h>
// Other includes
#include "utils.h"

#define G_DEBUG 0
#if G_DEBUG
    #define DEBUG_THRESH 0
    #define DEBUG_TX 0
    #define DEBUG_MATCH 0
    #define DEBUG_TX_DATA 1
    #define SCALE_ACC 0
#else
    #define DEBUG_THRESH 0
    #define DEBUG_MATCH 0
    #define DEBUG_TX 0
    #define DEBUG_TX_DATA 0
    #define SCALE_ACC 0
#endif


#define N_FFT 1024
#define N_RRC 1024

#define RBUF_BLOCKS 16 /// Num blocks in rolling buffer. MUST BE POWER OF 2.
```

```
52 #define WRAP_RBUF(x) (x & (RBUF_BLOCKS - 1)) /// Wrap buffer around
53
54 /**
55  * @brief MASDR Application Class
56  *
57  * This is the class for the MASDR Application. It contains all of the
58  * funcitonality that the platform will have. The platform consists of an SBC
59  * connected to a USRP SDR with an antenna.
60  */
61 class Masdr {
62 public:
63     /**
64      * @brief Initialize a Masdr object.
65      */
66     Masdr();
67
68     /**
69      * @brief Stop all functionality and destroy Masdr object.
70      */
71     ~Masdr();
72
73     /**
74      * @brief Update platform status
75      *
76      * Updates the location, heading, and stationarity.
77      */
78     void update_status();
79
80     /**
81      * @brief Handle software state transitions based on the current status.
82      *
83      * If the system is not currently idle, do not interrupt the current
84      * process. (This may need to change, but I don't see it being necessary)
85      */
86     void state_transition();
87
88     /**
89      * @brief Do any repetitive action associated with the current state.
90      *
91      * Gets called every loop and perfoms an action based on the current value
92      * of soft_status. In sample mode, this means starting a new recv buffer.
93      */
94     void repeat_action();
95
96     /**
97      * @brief Test the receive functionality.
98      *
99      * Test the functionality of the rx calling within the program.
100     */
101    void rx_test();
102
103    /**
104     * @brief Test the transmit functionality.
```

```
105      *
106      * Test the functionality of the tx calling within the program.
107      */
108     void tx_test();
109 
110     /**
111      * @brief Test the match filter amount.
112      */
113     void match_test();
114 
115     /**
116      * @brief Test the transmit data.
117      */
118     void transmit_data_test();
119 
120 
121 private:
122     /**
123      * @brief Initialize any peripherals being used
124      *
125      * This will be the GPS receiver and maybe an external memory device.
126      */
127     void initialize_peripherals();
128 
129     /**
130      * @brief Initialize the UHD interface to the SDR
131      *
132      * Initialize all components necessary for the interface to the USRP SDR
133      * using the UHD library.
134      */
135     void initialize_uhd();
136 
137     /**
138      * @brief Create a new thread to do the sampling.
139      */
140     void sample();
141 
142     /**
143      * @brief Detect if there's any energy detected on the bandwidth being measured.
144      *
145      * @param sig_in ///THIS IS NEEDED
146      * @param size ///THIS IS NEEDED
147      *
148      * @return ///THIS IS NEEDED
149      */
150     float energy_detection(std::complex<float> *sig_in, int size);
151 
152     /**
153      * @brief Transfer buffer to fft_in, and run FFT.
154      *
155      * @param ///THIS IS NEEDED
156      */
157     void run_fft(std::complex<float> *);
```

```
158
159     /**
160      * @brief Look for OFDM header.
161      */
162     float match_filt();
163
164     /**
165      * @brief Command the SDR to stop taking samples.
166      */
167     void stop_sampling();
168
169     /**
170      * @brief Start processing the collected samples.
171      */
172     void begin_processing();
173
174     /**
175      * @brief General transmission method.
176      *
177      * @param msg Pointer to packet to send.
178      * @param len Size of packet to be sent.
179      */
180     void transmit(std::complex<float> *msg, int len);
181
182     /**
183      * @brief Transmit data to ground station
184      *
185      * Transmit sampling location and directions for signals to ground station.
186      */
187     void transmit_data();
188
189     uhd::usrp::multi_usrp::sptr usrp;
190     std::complex<float> testbuf[RBUF_SIZE]; ///< Testing if structure is too big.
191     float rrcBuf[N_RRC]; ///< 4 samples per symbol.
192     uhd::rx_streamer::sptr rx_stream; ///< The UHD rx streamer
193     uhd::tx_streamer::sptr tx_stream; ///< The UHD tx streamer
194     uhd::rx_metadata_t md; ///< UHD Metadata
195     PhyStatus phy_status; ///< Physical status of the platform
196     SoftStatus soft_status; ///< The current stage of the software on the SBC
197     samp_block recv_buf[RBUF_BLOCKS]; ///< Rolling buffer of rcvd sample blocks
198     int rb_index; ///< Index of next insertion into recv_buff.
199     TransNode* trans_head; ///< Head node in linked list buffer for transmitting
200     TransNode* curr_trans_buf; ///< Last item in linked list.
201     fftw_plan fft_p; ///< FFTW Plan
202     fftw_complex ofdm_head[N_FFT]; //< Expected OFDM Header.
203     fftw_complex *fft_in, *fft_out; ///< Buffers for FFT.
204     bool process_done; ///< Set when data processing has completed
205     bool transmit_done; ///< Set when data transmission has completed
206     bool do_sample; ///< Disable to gracefully shutdown sampling.
207 };
208
209 #endif // __masdr_h__
```

B.2.4 masdr.cpp

```
// File: masdr.cpp
//
// MASDR Project 2016
// WPI MQP E-Project number:
// Members: Jonas Rogers
//          Kyle Piette
//          Max Li
//          Narut Akadejdechapanich
//          Scott Iwanicki
// Advisor: Professor Alex Wyglinski
// Sponsor: Gryphon Sensors

#include "masdr.h"
#include "utils.h"
#include "kalman_filt.h"
#include <iostream>
#include <fstream>
/*****************************************************************************/
Masdr::Masdr() {
    // Initialize software status
    process_done = false;
    transmit_done = false;
    soft_status = IDLE;

    // Initialize received sample buffer
    rb_index = 0;

    // Initialize FFTW
    fft_in = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * N_FFT);
    fft_out = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * N_FFT);
    fft_p = fftw_plan_dft_1d(N_FFT, fft_in, fft_out,
                             FFTW_FORWARD, FFTW_MEASURE);

    //Intialize OFDM Match Filter head
    //OFDM HEAD DETAILS:
        //Ignore first and last 174 buckets.
        //Each OFDM bucket is 16 fft buckets.
        //They are located at 254-270, 480-415, 640-615,864-880
    int i;
    for (i = 0; i < N_FFT; ++i) {
        int ratio = N_FFT/64; // Number of fft bins in an OFDM bin.

        // Account for the fact that only middle 52 OFDM bins are used.
        if (i < 6 * ratio || i > 58 * ratio) {
            ofdm_head[i][0] = 0;
            ofdm_head[i][1] = 0;
        }
        else if(i >= (6+6) * ratio && i < (6+7) * ratio) {
            ofdm_head[i][0] = 1;
            ofdm_head[i][1] = 1;
        }
```

```
52          else if(i >= (12+14) * ratio && i < (12+15) * ratio) {
53              ofdm_head[i][0] = 1;
54              ofdm_head[i][1] = 1;
55          }
56          else if(i >= (26+14) * ratio && i < (26+15) * ratio) {
57              ofdm_head[i][0] = 1;
58              ofdm_head[i][1] = 1;
59          }
60          else if(i >= (40+14) * ratio && i < (40 + 15) * ratio) {
61              ofdm_head[i][0] = 1;
62              ofdm_head[i][1] = 1;
63          }
64          else {
65              ofdm_head[i][0] = 0;
66              ofdm_head[i][1] = 0;
67          }
68      }
69
70      float time;
71      // Maybe internal freq should be 905e6,idk
72      int freq = 1e6/SPS; //Currently 1/2 symbol rate
73      float excess=0.2;
74      float Ts = 1/freq;
75      // float omega=2*PI*freq_tx; //2pif
76
77      for (i = 0; i < N_RRC;i++) {
78          if (i == N_RRC/2)
79              rrcBuf[i] = 1;
80          else{
81              time = (i - N_RRC/2)*Ts;
82              rrcBuf[i] = (sin(PI*time/Ts*(1-excess))+4*excess*time/Ts*cos(PI*time/Ts*(1+excess)))
83                          /(PI*time/Ts*(1-(4*excess*time/Ts)*(4*excess*time/Ts)));
84
85          }
86      }
87
88      initialize_peripherals();
89      initialize_uhd();
90      update_status();
91      do_sample = true;
92
93      boost::thread* s_thr = new boost::thread(boost::bind(&Masdr::sample, this));
94  }
95
96  /*************************************************************************/
97  Masdr::~Masdr() {
98      do_sample = false;
99      fftw_destroy_plan(fft_p);
100     fftw_free(fft_in);
101     fftw_free(fft_out);
102     delete trans_head;
103 }
104
```

```
105 /*****************************************************************************/
106 /************************INITIALIZATIONS**************************************/
107 /*****************************************************************************/
108 void Masdr::initialize_peripherals() {
109
110 }
111
112 /*****************************************************************************/
113 void Masdr::initialize_uhd() {
114     uhd::set_thread_priority_safe();
115
116     int rx_rate = 42e6;//To deal with the 20MHz bandwidth we have.
117     int tx_rate = 1e6; //4 samples per symbol at 700kHz
118     int master_rate = 42e6;
119     float freq_rx = 2.462e9; //Set rx frequency to 2.4 GHz //Set to 2.412 for channel 1
120
121     float freq_tx = 905e6; //set tx frequency
122     int gain = 50;// Default: 8dB
123     std::string rx_ant = "RX2"; //ant can be "TX/RX" or "RX2"
124     std::string tx_ant = "TX/RX"; //ant can be "TX/RX" or "RX2"
125     std::string wirefmt = "sc16"; //or sc8
126     int setup_time = 1.0; //sec setup
127
128     int rx_bw = 20e6; //11/16/16 MHLI: Should this be 10e6 and will it do half above half below?
129     int tx_bw = 300e3;
130
131     //Create USRP object
132     usrp = uhd::usrp::multi_usrp::make((std::string)"");
133     //Lock mboard clocks
134     usrp->set_clock_source("internal"); //internal, external, mimo
135     usrp->set_master_clock_rate(master_rate);
136     //set rates.
137     usrp->set_rx_rate(rx_rate);
138     usrp->set_tx_rate(tx_rate);
139
140     //Set frequencies.
141     uhd::tune_request_t tune_request_rx(freq_rx);
142     usrp->set_rx_freq(tune_request_rx);
143     uhd::tune_request_t tune_request_tx(freq_tx);
144     usrp->set_tx_freq(tune_request_tx);
145     //Set gain
146     usrp->set_rx_gain(gain);
147     usrp->set_tx_gain(gain);
148     //Set BW
149     usrp->set_rx_bandwidth(rx_bw);
150     usrp->set_tx_bandwidth(tx_bw);
151     //set the antennas
152     usrp->set_rx_antenna(rx_ant);
153     usrp->set_tx_antenna(tx_ant);
154
155     //allow for some setup time
156     boost::this_thread::sleep(boost::posix_time::seconds(setup_time));
157     //check Ref and LO Lock detect
```

```
158     check_locked_sensor(usrp->get_rx_sensor_names(0),
159                         "lo_locked",
160                         boost::bind(&uhd::usrp::multi_usrp::get_rx_sensor,
161                                     usrp, _1, 0),
162                         setup_time);
163 
164     //create streamers
165     //Initialize the format of memory (CPU format, wire format)
166     uhd::stream_args_t stream_args("fc32","sc16");
167     rx_stream = usrp->get_rx_stream(stream_args); //Can only be called once.
168     tx_stream = usrp->get_tx_stream(stream_args); //Can only be called once.
169 }
170 
171 /*************************************************************************/
172 /*************************STATE TRANSITIONS*******************************/
173 /*************************************************************************/
174 void Masdr::update_status() {
175     phy_status.heading = 0;
176     phy_status.is_stat_and_rot = false;
177     phy_status.location[0] = 0;
178     phy_status.location[1] = 0;
179     phy_status.location[2] = 0;
180 }
181 
182 /*************************************************************************/
183 void Masdr::state_transition() {
184     if (soft_status == IDLE) {
185         soft_status = PROCESS;
186         begin_processing();
187 
188     } else if (soft_status == PROCESS && process_done) {
189         soft_status = TRANSMIT;
190         process_done = false;
191         transmit_data();
192 
193     } else if (soft_status == TRANSMIT && transmit_done) {
194         soft_status = IDLE;
195         transmit_done = false;
196     }
197 }
198 
199 /*************************************************************************/
200 void Masdr::repeat_action() {
201     if (soft_status == SAMPLE) {
202         ;
203 
204     } else if (soft_status == PROCESS) {
205         ;
206 
207     } else if (soft_status == TRANSMIT) {
208         ;
209 
210     } else if (soft_status == IDLE) {
```

```
211         ;
212     }
213 }
214 
215 /*****************************************************************************/
216 /*********************************SAMPLE**************************************/
217 /*****************************************************************************/
218 
219 void Masdr::sample() {
220     // Create new sampling stream
221     uhd::stream_cmd_t start_strm_cmd(
222         uhd::stream_cmd_t::STREAM_MODE_START_CONTINUOUS);
223     start_strm_cmd.num_samps = size_t(0);
224     start_strm_cmd.stream_now = true;
225     start_strm_cmd.time_spec = uhd::time_spec_t(); // Holds the time.
226     rx_stream->issue_stream_cmd(start_strm_cmd);   // Initialize the stream
227     while (do_sample) {
228         rb_index = WRAP_RBUF(rb_index + 1);
229         recv_buf[rb_index].heading = phy_status.heading;
230         rx_stream->recv(recv_buf[rb_index].samples,
231                         RBUF_SIZE, md, 3.0, false);
232         boost::this_thread::interruption_point();
233     }
234     // Issue command to close stream
235     uhd::stream_cmd_t stop_strm_cmd(
236         uhd::stream_cmd_t::STREAM_MODE_STOP_CONTINUOUS);
237     rx_stream->issue_stream_cmd(stop_strm_cmd);
238 }
239 
240 /*****************************************************************************/
241 /*****************************PROCESSING**************************************/
242 /*****************************************************************************/
243 void Masdr::begin_processing() {
244     int i;
245     samp_block proc_buf[RBUF_BLOCKS/2];
246     float energy = 0;
247 
248     for (i = 0; i < RBUF_BLOCKS/2; ++i) {
249         proc_buf[i] = recv_buf[WRAP_RBUF(rb_index - RBUF_BLOCKS/2 + i)];
250     }
251     for (i = 0; i < RBUF_BLOCKS/2; ++i) {
252         energy += energy_detection(proc_buf[i].samples, RBUF_SIZE);
253     }
254     if(energy > THRESH_E) {
255         for (i = 0; i < RBUF_BLOCKS/2; ++i) {
256             run_fft(proc_buf[rb_index].samples);
257             float has_wifi =  match_filt();
258             if(has_wifi != -1)
259                 usrp->get_rx_sensor("rssi",0).to_real(); ///Not stored to anything rn
260         }
261     }
262 }
263 
```

```
264 /*******************************************************************************/
265 float Masdr::energy_detection(std::complex<float> *sig_in, int size) {
266     int i;
267     float acc = 0;
268     float max = 0;
269     float mag;
270 
271     for (i = 0; i < size; i++) {
272         mag = sqrt(sig_in[i].real() * sig_in[i].real()
273                    + sig_in[i].imag() * sig_in[i].imag());
274         acc += mag;
275         if (mag > max)
276             max = mag;
277     }
278 
279     if(DEBUG_THRESH) {
280         std::cout << max;
281         for (i = 0; i < (int)(mag*1000); i++){
282             std::cout << "#";
283         }
284         std::cout << std::endl;
285     }
286 
287     return max;
288 }
289 
290 /*******************************************************************************/
291 void Masdr::run_fft(std::complex<float> *buff_in) {
292     int i;
293     for(i = 0; i < N_FFT; ++i){
294         fft_in[i][0] = buff_in[i].real();
295         fft_in[i][1] = buff_in[i].imag();
296     }
297     fftw_execute(fft_p);
298 }
299 
300 /*******************************************************************************/
301 float Masdr::match_filt() {
302     int i;
303     int j;
304     float match_val[2] = {0,0};
305     float match_mag;
306     float re;
307     float im;
308 
309     for (i = 0; i < N_FFT; i++) {
310         re = fft_out[i][0] * ofdm_head[i][0] - fft_out[i][1] * ofdm_head[i][1];
311         im = fft_out[i][0] * ofdm_head[i][1] + fft_out[i][1] * ofdm_head[i][0];
312         match_val[0] += re;
313         match_val[1] += im;
314     }
315     match_mag = sqrt(match_val[0] * match_val[0]
316                      + match_val[1] * match_val[1]);
```

```
317     if (match_mag > THRESH_MATCH)
318         return match_mag;
319     else
320         return 0;
321 }
322
323 /*****************************************************************************/
324 /*****************************TRANSMISSION************************************/
325 /*****************************************************************************/
326 void Masdr::transmit_data() {
327
328     if (trans_head == NULL) {
329         std::cout << "No values to transmit" << std::endl;
330         return;
331     }
332
333     int i; // looping
334     int bias = 0; // compensating shift for adding more data
335     TransNode* trans_temp = trans_head;
336     std::complex<float> transmitBuffer[TBUF_SIZE];
337
338     // used to perform binary operations on floats
339     union {
340         float input;
341         int output;
342     } data;
343
344
345     std::cout << "Starting packaging" << std::endl;
346     while (trans_temp != NULL) {
347         //packing 33 start bits
348         for(i = 0; i < 33; i++) {
349             transmitBuffer[i+bias] = std::complex<float>(1,0);
350         }
351         bias += 33; // compensate for adding start bits
352
353         //packing gps data
354         data.input = trans_temp->gps[0];
355         for(i = 0; i < 32; i++) {
356             if ((data.output >> (31 - i)) & 1)
357                 transmitBuffer[i+bias] = std::complex<float>(1,0);
358             else
359                 transmitBuffer[i+bias] = std::complex<float>(-1,0);
360         }
361         bias += 32; // compensate for adding gps data 0
362
363         data.input = trans_temp->gps[1];
364         for(i = 0; i < 32; i++) {
365             if ((data.output >> (31 - i)) & 1)
366                 transmitBuffer[i+bias] = std::complex<float>(1,0);
367             else
368                 transmitBuffer[i+bias] = std::complex<float>(-1,0);
369         }
```

```
370         bias += 32; // compensate for adding gps data 1
371
372         //packing data
373         data.input = trans_temp->data;
374         for(i = 0; i < 32; i++) {
375             if ((data.output >> (31 - i)) & 1)
376                 transmitBuffer[i+bias] = std::complex<float>(1,0);
377             else
378                 transmitBuffer[i+bias] = std::complex<float>(-1,0);
379         }
380         bias += 32; // compensate for adding data
381
382         //packing 33 end bits
383         for(i = 0; i < 33; i++) {
384             transmitBuffer[i+bias] = std::complex<float>(-1,0);
385         }
386
387     std::ofstream ofs;
388     ofs.open ("/home/mqp/Results.bin", std::ofstream::out | std::ofstream::app);
389
390     ofs << transmitBuffer << std::endl;
391
392
393     ofs.close();
394
395     // uhd::tx_metadata_t md;
396     // md.start_of_burst = false;
397     // md.end_of_burst = false;
398
399     while(1) {
400         // tx_stream->send(transmitBuffer,TBUF_SIZE, md);
401
402     }
403     std::cout << "Done with transmit" << std::endl;
404 }
405 }
406
407 /*****************************************************************************/
408 void Masdr::transmit(std::complex<float> *msg, int len) {
409     uhd::tx_metadata_t md;
410     md.start_of_burst = false;
411     md.end_of_burst = false;
412     tx_stream->send(msg, len , md);
413 }
414
415 /*****************************************************************************/
416 /*********************************TESTS***************************************/
417 /*****************************************************************************/
418 void Masdr::rx_test() {
419     int i = 0;
420     int j = 0;
421     int numLoops; //Counter, to help
422     float accum;
```

```
423     float max_inBuf = 0;
424     float max_periodic = 0;
425     float max_total = 0;
426     float mag_squared;
427     double rssi;
428     int TX_Power = 0; //Transmit power of a wireless AP at 2.4 GHz
429     double dist_avg;
430     double calc_avg[1024];
431 
432     std::complex<float> testbuf[RBUF_SIZE];
433 
434     std::cout << "Entered rx_test" << std::endl;
435     sample();
436     std::cout << "Began sampling" << std::endl;
437     rx_stream->recv(testbuf, RBUF_SIZE, md, 3.0, false);
438     std::cout << "First Buff done" << std::endl;
439 
440     if(DEBUG_THRESH)
441         while (1){//(i < 5000) {
442         rx_stream->recv(testbuf, RBUF_SIZE, md, 3.0, false);
443         rssi = usrp->get_rx_sensor("rssi",0).to_real();
444         if(rssi > -78){
445             calc_avg[i++] = pow(10,((rssi)/-20));
446         }
447 
448         if (i > 1023) {
449             for (j = 0; j < 1024; j++) {
450                 dist_avg += calc_avg[j];
451             }
452             std::cout << dist_avg/1024 << std::endl;
453             dist_avg = 0;
454             i = 0;
455         }
456 
457         //std::cout << energy_detection(testbuf, RBUF_SIZE) << std::endl;
458     }
459     do_sample = false;
460     std::cout << "Stopped sampling" << std::endl;
461     std::cout << "RX test done." << std::endl << std::endl;
462 }
463 
464 /**********************************************************************/
465 void Masdr::tx_test() {
466     int i; //Counter, to help test
467     std::complex<float> testbuf[100];
468 
469     std::cout << "Entered tx_test" << std::endl;
470     //Initialize test buffer.
471     for (i = 0; i < 100; ++i) {
472         testbuf[i] = std::complex<float> (1,0);
473     }
474 
475     uhd::tx_metadata_t md;
```

```
476     md.start_of_burst = false;
477     md.end_of_burst = false;
478 
479     std::cout << "Began transmit" << std::endl;
480     tx_stream->send(testbuf, 100, md);
481     std::cout << "First Buff done" << std::endl;
482 
483     i = 0;
484     while (1) {
485         tx_stream->send(testbuf, 100, md);
486         ++i;
487     }
488 
489     std::cout << "Stopped transmit" << std::endl;
490     std::cout << "Tx test done." <<std::endl<<std::endl;
491 }
492 
493 /*********************************************************************/
494 void Masdr::match_test() {
495     //Test match filt stuff.
496     float test_val;
497     int i;
498 
499     while(1) {
500         for (int j = 0; j < RBUF_BLOCKS/2; ++j) {
501             run_fft(recv_buf[WRAP_RBUF(rb_index - RBUF_BLOCKS/2 + j)].samples);
502 
503             test_val = match_filt();
504             //std::cout << test_val;
505             for (i = 0; i < (int)test_val*5; ++i)
506                 std::cout << '#';
507             std::cout << std::endl;
508         }
509     }
510 
511     std::cout << "Match filter test done." << std::endl << std::endl;
512 }
513 
514 /*********************************************************************/
515 void Masdr::transmit_data_test() {
516     int i;
517     std::cout << "In tx data test" << std::endl;
518     transmit_data();
519 }
520 
521 /*********************************************************************/
522 int UHD_SAFE_MAIN(int argc, char *argv[]) {
523     std::signal(SIGINT, handle_sigint);
524     Masdr masdr;
525 
526     if (G_DEBUG) {
527         if (DEBUG_THRESH) masdr.rx_test();
528         if (DEBUG_TX) masdr.tx_test();
```

```
529         if (DEBUG_MATCH) masdr.match_test();
530         if (DEBUG_TX_DATA) masdr.transmit_data_test();
531     }
532 
533     else {
534         while(1) { //!stop_signal_called)
535             masdr.update_status();
536             masdr.state_transition();
537             masdr.repeat_action();
538         }
539     }
540 
541     return EXIT_SUCCESS;
542 }
```

Appendix C

Python Code

C.1 Map Generation Code

```
1
2 from math import sqrt
3
4 # Read in measurements from file
5 measurements = []
6 with open('gps_vals_urban.txt', 'r') as gps_vals:
7     gps = gps_vals.readlines()
8     with open('rss_vals_urban.txt', 'r') as rss_vals:
9         rss = rss_vals.readlines()
10        rss_scale = len(rss)/len(gps)
11        offset_b = 900
12        offset_e = 200
13        reduction = 90
14        for i in range((len(gps) - offset_b - offset_e) / reduction):
15            measurements.append((float(str(gps[offset_b+reduction*i]).split()[0]),
16                                 float(str(gps[offset_b+reduction*i]).split()[1]),
17                                 float(rss[(offset_b+reduction*i)*rss_scale].strip('\n,'))))
18
19 altitude = 20.0 # in meters
20 # (Lat, Long, RSSI value)
21 # measurements = [(42.274744, -71.8084369, -84.3),
22 #                 (42.275376, -71.8085379, -85.3),
23 #                 (42.275342, -71.8075235, -85.9)]
24
25 # Distance calculation
26 # Pythagorean theorem to eliminate altitude
27 rss_dist_meas = [(meas[0], meas[1],
28                 (10**((meas[2]/-20))/100) / 25)
29                 for meas in measurements]
30
```

```
# Populate js list of points to plot
circle = '    {point}: {{center: {{lat: {lat}, lng: {lng}}}, distance: {dist}}}'
circles = [circle.format(point=index,
                         lat=rss_dist_meas[index][0],
                         lng=rss_dist_meas[index][1],
                         dist=rss_dist_meas[index][2],)
           for index in xrange(len(rss_dist_meas))]
mapjs = ',\n'.join(circles)

# Fill in template with data
with open('map_template.html', 'r') as temp:
    html_map = temp.read()
html_map = html_map.format(points=mapjs,
                           center_lat=rss_dist_meas[0][0],
                           center_lng=rss_dist_meas[0][1])
with open('masdr_localization.html', 'w') as f:
    f.write(html_map)
```

Appendix D

MATLAB Code

D.1 Energy Detection Example

```
clear all;
close all;
N = 10000; % N is length of Signal
Nbins = 1024;
%Set Threshold to check if signal is there.
thresh = 200;

f = 500e3;
Fs = 5e6;
t = (1:N)* 1/Fs;

noise = randn(1,N);
sig = cos(2*pi*f * t);

% Do FFT on time, noise.
fft_noise = fft(noise, Nbins);
fft_sig = fft(sig, Nbins);

mag_noise = abs(fft_noise);
mag_sig = abs(fft_sig);

% Plot FFT results
figure(1);
% Hist values in bins, randn.
plot(-(Nbins-1)/2:(Nbins-1)/2,mag_noise,-(Nbins-1)/2:(Nbins-1)/2,thresh*ones(1,Nbins));
title('FFT Bin Values, Noise');
axis([-512 512 0 250]);

figure(2);
% Hist values in bins, cos.
```

```
31 plot(-(Nbins-1)/2:(Nbins-1)/2,mag_sig,-(Nbins-1)/2:(Nbins-1)/2,thresh*ones(1,Nbins));
32 axis([-512 512 0 250]);
33 title('FFT Bin Values, Signal');
34
35
36 % Determine if either value passes threshold.
37 noise_passes_thresh = (max(mag_noise)>thresh)
38 sig_passes_thresh = (max(mag_sig)>thresh)
```

D.2 Matched Filter Example

```
clear all;
close all;

%Seed to generate expected value (so threshold is always applicable).
rng(500);
%Size of random signal, representing noise
N=400;
%pick thresh to separate matched result to unmatched result.
thresh=20;

%Example packet to look for
toMatch = [1 -1 -1 -1 1 1 -1 1 1 -1 -1 -1 -1 -1 1 1 1 -1 1 1 1 1 -1 -1 1];
%Reverse, so a filter operation will instead correlate. Shifted back to
%positive to be causal.
reverseMatch = toMatch(end:-1:1);

%Generate random signal.
noise = randn(1,N);
%make signal with noise and
noise_signal = noise;

%Add expected packet to signal.
for i=50:length(toMatch) + 49
    noise_signal(i) =  noise(i)+toMatch(i-49);
end

figure(1);
plot(noise_signal);
title('Noisy Signal With Match Header');

%filter noise_signal
filt_sig= filter(reverseMatch,1, noise_signal);
figure(2);
plot(1:length(filt_sig),filt_sig,1:length(filt_sig),thresh*ones(length(filt_sig)));
title('Match Filtered Signal');
```

D.3 GPS Kalman Filter Simulation

```
clear all;
close all;

numTrials = 10000;
dt = 0.01;

%State transition Matrix
F = [1 0 dt 0; 0 1 0 dt; 0 0 1 0; 0 0 0 1];
%Initialize state to 0;
x=zeros(4,numTrials+1);
x(:,1) = [0;0;0;0;];

%Initialize state covariance matrix
var_dx = 2;
var_dy = 2;
var_vx = 4;
var_vy = 4;
p= [var_dx 0 0 0; 0 var_dy 0 0; 0 0 var_vx 0; 0 0 0 var_vy];

%initialize H
H = [1 0 0 0; 0 1 0 0; 0 0 1 0; 0 0 0 1];

%Initialize noise covariance matrices
sd_q = 0.9;
sd_r = 0.5;
%Q is piecewise white noise.
Q = sd_q*[1/4*dt^4 0 0.5*dt^3 0; 0 1/4*dt^4 0 0.5*dt^3; 0.5*dt^3 0 dt^2 0; 0 0.5*dt^3 0 dt^2];
%R is piecwise white noise
R = sd_r * [1/4*dt^4 0 0.5*dt^3 0; 0 1/4*dt^4 0 0.5*dt^3; 0.5*dt^3 0 dt^2 0; 0 0.5*dt^3 0 dt^2];

%Initialize input
M = zeros(4, numTrials + 1);
for i = 1:numTrials+1
    xrand = rand()/2;
    yrand = rand()/2;
    if i == 1
        M(:,i) = [i+xrand i+yrand 1 1];
    else
        M(:,i) = [i+xrand i+yrand 0 0];
        M(3,i) = M(1,i)-M(1,i-1); %Velocity isn't directly measured.
        M(4,i) = M(2,i)-M(1,i-1);
    end
end

for i = 1:numTrials
    %predict Step
    x(:,i) = F*x(:,i);
    p = F*p*F.'+ Q;
    %correct step
    K = p*H.'\(H*p*H.' + R);
    x(:,i+1) = x(:,i) + K*(M(:,i)-H*x(:,i));
```

```
    p = (eye(4)-K*H)*p;
end
x(:,numTrials)

figure(1);
plot(0:numTrials,x(1,:),1:numTrials+1,M(1,:));
axis([0 numTrials -1 numTrials] );
title('X Position Prediction Over Time');
figure(2);
plot(0:numTrials,x(2,:),1:numTrials+1,M(2,:));
axis([0 numTrials -1 numTrials] );
title('Y Position Prediction Over Time');
figure(3);
plot(0:numTrials,x(3,:),1:numTrials+1,M(3,:));
axis([0 numTrials -2 2] );
title('X Velocity Prediction Over Time');figure(1);
figure(4);
plot(0:numTrials,x(4,:),1:numTrials+1,M(4,:));
axis([0 numTrials -2 2] );
title('Y Velocity Prediction Over Time');

%figure(2);
```

D.4 Post-Processing Script

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% data_manip_masdr.m: by Max Li, 2016
% Reads direct data received from USRP
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
clear all;
close all;

Ts = 1;
%Ts=42e6;
%Read in samples from file
fname = 'test_wyg_data_12-11.dat';
fid = fopen(fname,'rb');
tmp = fread(fid,'float');
samples = zeros(length(tmp)/2,2);
samples(:,1) = tmp(1:2:end);
samples(:,2) = tmp(2:2:end);
length_temp = length(tmp);
clear tmp;

% Initialize counters
count_sig = 1;
count_match = 1;
count_gps_x = 1;
count_gps_y = 1;
count_gps_z = 1;
count_rss = 1;
count_samp = 1;

% Initialize buffers storing various information saved
match_sig = zeros(nnz(samples(:,2)==1000),1);
gps_x = zeros(nnz(samples(:,2)==2000),1);
gps_y = zeros(nnz(samples(:,2)==3000),1);
gps_z = zeros(nnz(samples(:,2)==4000),1);
rss_val = zeros(nnz(samples(:,2)==5000),1);
samp_act = zeros(length_temp/2 - nnz(samples(:,2)==4000)...
                - nnz(samples(:,2)==3000) - nnz(samples(:,2)==2000)...
                - nnz(samples(:,2)==1000),2);

% Separate Match values, gps values from sampled signals.
for i = 1:length(samples)
    if samples(i,2) == 1000
        match_sig(count_match) = samples(i,1);
        count_match = count_match + 1;
    else if samples(i,2) == 2000
            gps_x(count_gps_x) = samples(i,1);
            count_gps_x = count_gps_x + 1;
        else if samples(i,2) == 3000
                gps_y(count_gps_y) = samples(i,1);
                count_gps_y = count_gps_y + 1;
            else if samples(i,2) == 4000
                    gps_z(count_gps_z) = samples(i,1);
```

```
                    count_gps_z = count_gps_z + 1;
                else if samples(i,2) == 5000
                        rss_val(count_rss) = samples(i,1);
                        count_rss = count_rss + 1;
                    else
                        samp_act(i) = samples(i);
                        count_samp = count_samp + 1;
                    end
                end
            end
        end
    end
end
clear samples;
% mean(samples(:,1));
% mean(samp_act(:,1));
%
% figure(1);
% plot(1:Ts:(length(samp_act))/Ts,samp_act(:,1));
% title('Sampled Data');

figure(2);
plot(1:Ts:(count_match-1)/Ts,match_sig);
title('Matched Filter Values');

figure(3);
plot(1:Ts:(count_rss-1)/Ts,rss_val);
title('RSS Values');
%
% %fs = 640000;
```