2018

Leveraging Software-Defined Radio for a Scalable Wide-band Wireless Channel Measurement System

James Jamison
University of Vermont

Leveraging Software-Defined Radio for a Scalable Wide-band Wireless Channel Measurement System

A Thesis Presented

by

James A. Jamison III

to

The Faculty of the Graduate College

of

The University of Vermont

In Partial Fulfillment of the Requirements
for the Degree of Master of Science
Specializing in Electrical Engineering

October, 2018

Defense Date: August 24, 2018
Thesis Examination Committee:

Jeff Frolik, Ph.D., Advisor
Safwan Wshah, Ph.D., Chairperson
Tian Xia, Ph.D.
Cynthia J. Forehand, Ph.D., Dean of Graduate College

Abstract

Wireless channel characterization is important for determining both the requirements for a wireless system and its resulting reliability. Wireless systems are becoming ever more pervasive and thus are expected to operate in increasingly more cluttered environments. While these devices may be fixed in location, the channel is still far from ideal due to multipath. Under such conditions, it is desirable to have a means of taking wireless channel measurements in a low-cost and distributed manner, which is not always possible using typical channel measurement equipment.

This thesis leverages a software-defined radio (SDR) platform to perform wide-band wireless channel measurements. Specifically, the system can characterize the scalar frequency response of a wireless channel in a distributed manner and provides measurements with an average mean-squared error of 0.018 % σ and a median error not exceeding 0.631 dB when compared to measurements taken with a Vector Network Analyzer. This accuracy holds true in a highly multipath environment, with a measurement range of $\sim$ 40 dB. The system is also capable of scaling to multiple wireless links which will be measured simultaneously (up to three links are demonstrated). After validating the measurement system, a measurement campaign is undertook using the system in a highly multipath environment to demonstrate a possible application of the system.

Citations

Material from this thesis has been published in the following form:

Jamison, J., Frolik, J.. (2018). *A Software-Defined Radio Approach to Multi-Link Channel Characterization.* 2018 IEEE 19th Wireless and Microwave Technology Conference (WAMICON), Sand Key, FL, 2018, pp. 1-4.

Acknowledgements

First I would like to thank my parents for their unconditional support throughout my life, especially during my college career. Without the opportunities they have provided me with I would not be in this position writing a graduate thesis. I would also like to thank my girlfriend Holley for her support over the last year as I struggled with classes and thesis writing. I would like to thank Dr. Tian Xia and Dr. Safwan Wshah for taking the time to serve on my thesis committee.

Lastly, I would like to thank my advisor Dr. Jeff Frolik for everything that he has done to help me prepare for my engineering career during my times as an undergraduate and graduate student; he is a fantastic educator and mentor. Working with him has shaped my interests and led to the opportunities I have before me today.

The work for this thesis was supported by the Vermont Space Grant Consortium under NASA Grant and Cooperative Agreement NNX15AP86H and by NSF grant EECS-1508907.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

In recent years, the Internet of Things (IoT) has become a popular buzzword in the electronics field, specifically in communications. IoT refers to the wireless connection of everyday electronic devices to each other and to the Internet [1]. An early example of an IoT system is the Wireless Sensor Network (WSN). As its name suggests, a WSN is a network of wirelessly connected sensors deployed to collect data on an environment in a distributed fashion. WSN have many benefits over their wired counterparts, but problems arise in trying to operate these networks in some harsh communications environments.

Such environments are the cluttered and reflective ones present in Machine-to-Machine (M2M) communication scenarios. The M2M communications market, which includes applications such as vehicular communications for self driving cars and traffic control, is expected to grow from \$47.9B in 2017 to \$199.6B by 2022 [2]. The reflective environments expected for these applications have been shown to exhibit

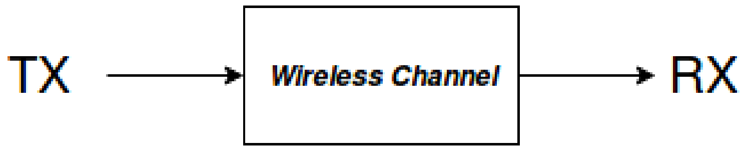


Figure 1.1: Flow graph of a generic wireless channel.

severe frequency selective fading which is difficult to operate a wireless device in. Other highly reflective cluttered environments will be seen by systems operating on factory floors and in air-frames (e.g., transport helicopters [3] and commercial airplanes [4]). Due to the difficulty of operating wireless devices in these environments, signal propagation measurements to be used to characterize the channels can be very useful.

1.2 Problem Statement

The motivation of this work stems from a desire to characterize wireless channels, displayed in Fig. 1.1, for new IoT applications. Wireless channel characterization allows one to gain insight into the types of wireless devices that may operate effectively in that environment and the resulting reliability of their operations. Measurement campaigns have been undertook in the past attempting to characterize these environments [3, 4, 5, 6] and work has also been done on emulating these environments in a laboratory setting [7, 8] but the methods used to measure these channels is often pricey and difficult.

One can obtain the characteristics of a wireless channel in either the time or frequency domain and methods exist for obtaining both the impulse response (time-domain information) and frequency response of a wireless channel. Typically the impulse response is obtained through direct RF pulse measurements or spread spectrum sliding correlator systems [9] (these systems will be discussed in Section 2.3.1). These

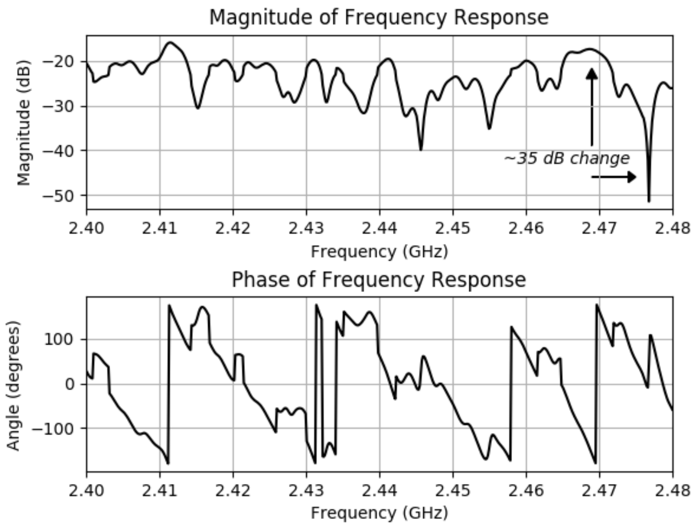


Figure 1.2: An example S_{21} plot for a measurement in a highly reflective wireless channel over the 2.4 GHz ISM Band (2.40 GHz - 2.48 GHz).

methods usually employ non-coherent transmitter and receiver systems and transmit wide-band pulses. These wide-band pulses require a measurement system with a large bandwidth and can therefore sometimes be very expensive. In order to obtain the frequency domain information from the impulse response through an inverse Fourier transform, the system must be coherent (i.e. the transmitter and receiver must have synchronized clocks) which can be difficult to implement in practice.

In some applications, the frequency response can be more valuable than time domain information (this argument is made for highly reflective environments in Section 2.3.2). This measurement is typically obtained using a Vector Network Analyzer (VNA) which provides a measurement known as an S_{21} measurement which gives the magnitude and phase of the received signal relative to a transmitted signal at incremental frequencies (an example in Fig. 1.2). VNAs are extremely expensive and are coherent measurement systems, meaning the transmitter and receiver must be connected to the main system. This can cause issues when measuring wireless links as cables need to be run across the environment, causing difficulty in set-up and possible interference with the measurement.

Relatively recently, SDR has been proposed as a way of providing these channel measurements at a fraction of the cost as traditional means (prior work discussed in Section 3.4), but even these systems have some shortcomings for this application. This thesis presents a SDR based channel measurement system developed specifically for the distributed measurement of the scalar frequency response of wireless links in highly reflective environments.

1.3 Contributions

The work in this thesis proposes a channel measurement system based entirely in software-defined radio (SDR). The main contributions of this work are three-fold:

1. *Development of a novel distributed channel measurement technique based in software defined radio.*

   This measurement system utilizes software defined radios, and a "chirped" tone, to measure the scalar frequency response of a wireless channel. The system is detailed in Section 4.1.1.

2. *Development of a algorithm capable of synchronizing multiple software-defined radios over a bandwidth larger than their operational bandwidth for multiple distributed measurement links.*

   The system utilizes radios with an operational bandwidth (10 MHz) smaller than the measurement bandwidth (80 MHz) in order to reduce the cost of the system. Because of this smaller operational bandwidth, the operating frequencies of the SDRs needs to be changed over the course of a measurement. Furthermore, these operating frequencies must be changed between distributed "ends" of the system simultaneously. A novel algorithm is developed to achieve this and presented in Section 4.1.2. In Section 4.1.5, this algorithm is extended from a single measurement link to N measurement links using Time Division Multiple Access (TDMA) techniques.

3. *Demonstration of channel measurement system to validate system accuracy.* Measurements are taken using the proposed measurement system in a highly reflective environment and compared to measurements taken using a Vector Network Analyzer (VNA) in Section 4.2. It is found that the system is able to measure with a mean-squared error of 0.018% σ and a median error which does not exceed 0.631 dB when compared to VNA measurements. These measurements are taken over a 80 MHz bandwidth on channels whose signal strength extends $\sim$40 dB; both single link and multi-link measurements are demonstrated.

There exists limited prior work in SDR based channel measurement systems, a summary of which is presented in Section 3.4, but the proposed system significantly extends any existing systems developed for similar uses as per the contributions above. In addition to the contributions of the system itself, a measurement campaign is undertook in Chapter 5 which presents a realistic application of the system as well as adding to the literature on wireless channel measurements in highly cluttered environments.

1.4 Thesis Outline

This thesis is organized as follows. First, models to mathematically describe wireless channels and existing methods of empirically measuring channels are both discussed in Chapter 2. Chapter 3 gives an introduction to what software-defined radio is, how it works, and details the existing work on software-defined radio based channel measurement devices. Chapter 4 introduces the system being proposed by this thesis and provides validating measurements comparing the system's measurements to that of a Vector Network Analyzer. Lastly, in Chapter 5 a possible application of the proposed system is discussed and Chapter 6 concludes this thesis with a discussion of future work and some final words.

Chapter 2

Wireless Propagation

As this thesis aims to develop novel approaches for channel characterization, it is worthwhile to examine the existing mathematical models used to describe wireless channels. Modeling wireless channels mathematically allows for the easy comparison of one channel to another, and for the computation of signal propagation loss "severity". In this chapter, mathematical models of wireless signal propagation will be examined by first introducing the propagation phenomenon and exploring large-scale propagation, followed by more closely looking at small-scale fading models. Finally we will look at existing methods of collecting the data needed to characterize wireless channels.

2.1 Large-Scale Propagation

Wireless signal propagation is broken into two categories for analysis: large-scale signal propagation and small-scale signal propagation. As the name implies, large-scale signal propagation models concern propagation effects due to large changes in position, frequency or time, while small-scale propagation models concern propagation effects due to small changes. In other words, large-scale models characterize changes in *average* signal strength, while small-scale models characterize deviations from this average [9]. This thesis is concerned more with propagation on a small-scale than a large-scale, but large-scale propagation models are presented in this section for completeness.

Large scale propagation in its simplest form occurs in free space (e.g., satellite-to-satellite communications), and can be modeled by the Friis free space equation:

$$P_r(d) = \frac{P_t G_t G_r \lambda^2 L}{(4\pi)^2 d^2} \tag{2.1}$$

where P_t is transmitted power, P_r is the received power, G_t and G$_r$ are the gains on the transmit and receive antenna respectively, d is the distance from transmitter to receiver, λ is the signal wavelength, and L ($0 \leq L \leq 1$) is the system loss factor (not related to propagation, a value of L=1 indicates no loss in system hardware). In practice, path loss (PL) is often expressed in dB, as shown below:

$$PL(dB) = 10log\frac{P_t}{P_r} = -10log\frac{G_t G_r \lambda^2}{(4\pi)^2 d^2} \tag{2.2}$$

Using Eq. 2.2, average signal power loss can be calculated in free space scenarios. The Friis equation (2.1) shows us that received power decreases as the square of the distance from receiver to transmitter. It is worth noting that these equations are only valid for values of d which are within the far-field of the transmitting antenna, or the Fraunhofer distance (d_f), displayed below:

$$d_f = \frac{2D^2}{\lambda} \tag{2.3}$$

where D is the largest linear dimension of the antenna.

Eq. 2.2 gives insight into path loss relative to transmitter receiver separation in free space. From [9], the mean path loss is expressed by the following proportion:

$$\overline{PL}(d) \propto \left(\frac{d}{d_o}\right)^n \tag{2.4}$$

where n is the path loss exponent, and d_o is the "close-in reference distance" (often set to 1m for convenience). Eq. 2.4 expressed in dB, with the addition of a random variable is known as the log-normal shadowing model, shown below:

$$PL(d) = \overline{PL}(d) + X_\sigma = PL(d_o) + 10nlog(\frac{d}{d_o}) + X_\sigma \tag{2.5}$$

where $X_\sigma \sim N(0, \sigma)$ (note: σ is expressed in dB). The log-normal shadowing model will accurately measure the average signal strength of a signal transmitted over a distance. The values of n and σ will vary from one environment to the next, and must be estimated based on empirical data from similar environments when using this model.

2.2 Small-Scale Propagation

Of more interest to this thesis is the variations around the average path loss value that occurs as a result of small changes in either frequency, position or time. Historically, small-scale propagation pertained to changes in time as these models were originally developed for mobile systems, such as those present in cellular telephone networks [10]. With the rising interest in the internet of things (IoT) and machine-to-machine communications (M2M), it is worth considering temporally static environments where small changes in position and frequency can cause large changes in signal strength. For example, > 30 dB variations have been demonstrated for positional changes less than $\lambda/10$ in the 5.7 GHz ISM band [5] and similar variations were found for small changes in frequency in the 2.4 GHz ISM band [3]. Both of the aforementioned measurement campaigns took place in highly reflective cluttered environments which exhibited severe multipath, and therefore extreme frequency selective fading.

In this section, the statistical distributions used to model this behavior will be introduced as they model environments that may be similar to those that Internet-of-Things (IoT) and machine-to-machine (M2M) systems will be deployed in. Namely the Ricean, Rayleigh, and hyper-Rayleigh models used to model small-scale fading are discussed. These models are introduced now as they are referenced heavily throughout this thesis.

Small-scale fading is a result of multipath components in the communication channel adding together at the receiver. The multipath components travel along different paths to the receiver than the line-of-sight (LOS) signal component and as such reach the receiver at different times (i.e., phases). When these signal components add together at the receiver out of phase, they will sometimes add constructively and other times add deconstructively, causing these quick variations in signal strength. In these cases, it is often more practical to consider statistical models based on empirical data, as opposed to physical deterministic models, as the environmental reflections can be quite complex and relatively random.

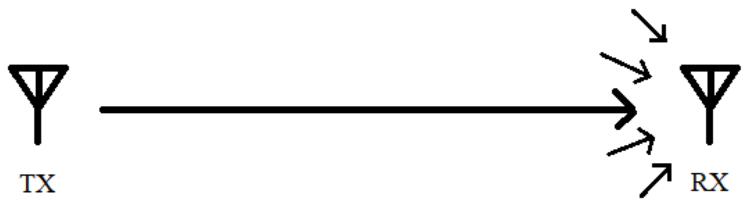


Figure 2.1: A sketch of a Ricean communication channel. The dominant LOS component is evident as the darker signal component extending directly from TX to RX.

In general, the following derivations are based on the mathematical description of the summation of constant-amplitude waves with "N" independently identically distributed (I.I.D.) phases:

$$\tilde{V} = \sum_{i=1}^{N} V_i e^{j\phi_i} \tag{2.6}$$

where $\tilde{V}$ is the complex baseband voltage, the V_i's are the amplitudes of the multipath waves and the ϕ's are the phases of the multipath phases. It is also worth noting that $Re\{V_i\} \sim N(0, \sigma)$ and $Im\{V_i\} \sim N(0, \sigma)$ due to the central limit theorem [11]. Unless otherwise noted this result as well as the following derivations come from [12].

2.2.1 Ricean Fading

The Ricean distribution is used to model communication scenarios with a dominant line-of-sight (LOS) signal component, interacting with multiple multipath components. This model is used in multipath environments where the LOS component is relatively strong compared to multipath components, as seen in Fig. 2.1. In this case, the summation from Eq. 2.6 is used to sum the multipath components, and this is added to the LOS signal component to compute the received signal:

$$\tilde{V}_{RX} = V_1 e^{j\phi_1} + \sum_{i=2}^{N} V_i e^{j\phi_i} \tag{2.7}$$

where $\tilde{V}_{RX}$ is the complex valued received signal voltage, V_1 is the amplitude of the complex valued dominant (typically LOS) signal component with phase ϕ_1, and there are L multipath components with amplitude V_i and phase ϕ_i. The multipath components in this lumped term are commonly referred to as the diffuse components. The probability density function (PDF) of the Ricean case (derived in [12]) is:

$$f_R(r) = \frac{r}{\sigma^2} e^{(\frac{-r^2 - V_1^2}{2\sigma^2})} I_0(\frac{rV_1}{\sigma^2}) \tag{2.8}$$

where r is the envelope amplitude, σ is the standard deviation of envelope voltage, and $I_0(\cdot)$ is the zeroth-order modified Bessel function. A common metric for determining the severity of various Ricean fading channels is through the Ricean K-factor. The K-factor is the ratio of the power of the dominant component to the power of the diffuse multipath components:

$$K = \frac{V_1^2}{2\sigma^2} \tag{2.9}$$

It can be seen from Eq. 2.9 that an increase in the K-factor would indicate a stronger dominant component. In practice the K-factor is often expressed in dB (i.e. $10\log_{10}$ of Eq. 2.9).

Figure 2.2: A sketch of a Rayleigh communication channel. The dominant LOS component is no longer discernible from the multipath components.

2.2.2 Rayleigh Fading

The Rayleigh distribution is used as a worst-case model in cellular communications in which there is no single dominant signal component (i.e., the LOS is either missing or not distinguishable from the multipath). In other words, $K = 0$ (-∞ dB) in the Rayleigh distribution (an example Rayleigh channel is provided in Fig. 2.2).

While this type of fading physically exhibits more deep fades than the Ricean, it is simpler mathematically as the lumped diffuse component can be treated as a single random variable.

$$\tilde{V}_{RX} = \sum_{i=1}^{N} V_i e^{j\phi_i} \tag{2.10}$$

As in the Ricean case, the lumped component is a sum of I.I.D. complex variables that are normally distributed. From the central limit theory [11] it is known that the sum of I.I.D. random variables approach the Gaussian distribution. Furthermore, the joint probability of two random Gaussian variables (i.e., the real and imaginary parts of the diffuse component) yields a Rayleigh probability density function of the signal's envelope [11], displayed below:

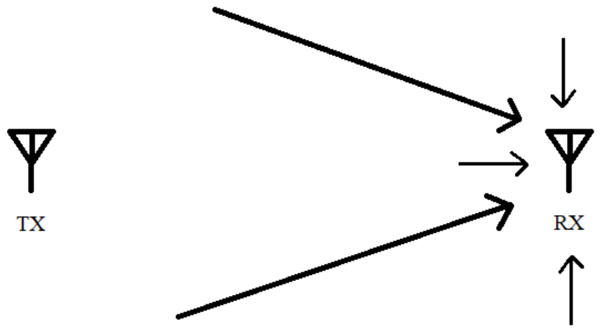


Figure 2.3: A sketch of a hyper-Rayleigh communication channel. There is more than one dominant component present.

$$f_R(r) = \frac{r}{\sigma^2} e^{\frac{-r^2}{2\sigma^2}} \tag{2.11}$$

where r is the envelope amplitude, and σ is the standard deviation of envelope voltage. This Rayleigh PDF is often used as a benchmark when examining empirical data because it can be inferred that when data exhibits this distribution, the line-of-sight component has been lost. This condition was thought to be the worse case scenario in a typical multipath communication channel.

2.2.3 Hyper-Rayleigh Fading

Up until about a decade ago, fading which was more statistically severe than Rayleigh had been theorized but not measured empirically. In [6], channels exhibiting fading more statistically severe than Rayleigh were measured, and hyper-Rayleigh fading was a term proposed to include any fading more severe than Rayleigh. In this scenario, displayed in Fig. 2.3, multiple signal components of near equal strength are present at the receiver and they can no longer be lumped together into a single diffuse component. This only happens in extremely reflective environments which are capable of producing these severe signal reflections (e.g., a metallic air frame).

In this case the received signal may look more like:

$$\tilde{V}_{RX} = V_1 e^{j\phi_1} + V_2 e^{j\phi_2} + \sum_{i=3}^{N} V_i e^{j\phi_i} \tag{2.12}$$

where there are multiple dominant (called secular) components summed with the usual lumped diffuse component (in Eq. 2.12 there are two secular components, but there could in theory be more). There exist rather mathematically involved models, and accompanying PDFs, such as a the two-wave with diffuse power (TWDP) and three-wave fading scenarios [12], to model these channels. But for our purposes it suffices to refer to any channel which exhibits fading more statistically severe than Rayleigh as "hyper-Rayleigh".

2.3 Wireless Channel Measurement Techniques

The models presented so far in this chapter give scientists and engineers intuition into the behavior of wireless signals, but they are especially useful when they can be used to describe signal propagation in a physical environment. Typically this is done by empirically measuring a wireless channel, and then using this data to determine the behavior of the wireless signals in the channel (i.e., if the channel exhibits Ricean/Rayleigh/hyper-Rayleigh small-scale fading). This is important not only for determining the requirements of a wireless system operating in such a channel, but also for determining the resulting reliability of the system. As the aim of this thesis is to propose a novel method for empirically collecting channel data, this section

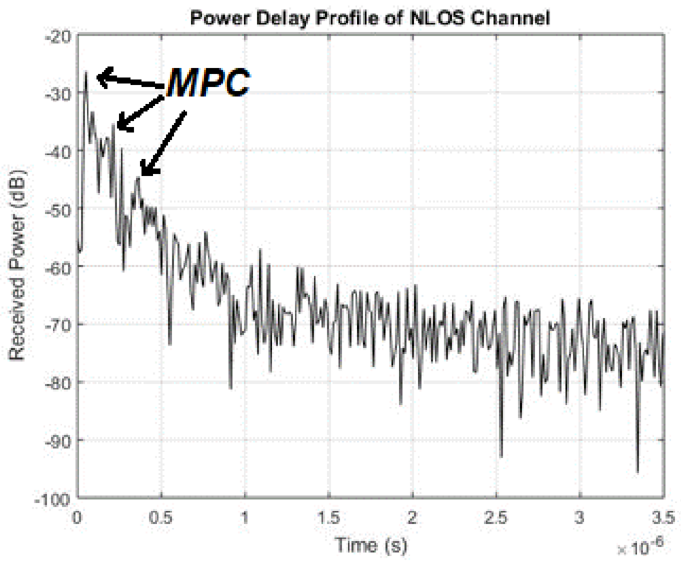


Figure 2.4: An example power delay profile (PDP) for a non-line of sight (NLOS) multipath channel. The received signal components (labeled as MPC for multipath components) are labeled.

presents existing methods of wireless channel measurement. Channel measurement can be done either in the time domain, or the frequency domain; both of these methods will be discussed.

2.3.1 Time Domain Channel Sounding

Channel measurement in the time domain is typically called channel sounding. The goal of channel sounding is to obtain the channel impulse response (CIR) or power delay profile (PDP). As per the basic principles of system theory, obtaining the impulse response of a channel would allow one to calculate the out response of that channel to any input signal, which can be very useful. An example PDP for a non-line of sight measurement in a highly multipath wireless channel is displayed in Fig. 2.4. The multipath components can be seen as delayed spikes in the PDP. The direct radio-frequency (RF) pulse system is the simplest form of measuring the channel impulse response.

The direct RF pulse channel sounding method is presented in detail in [9], and demonstrated in [13] and [14]. It is a relatively simple channel sounding method, and involves transmitting a narrow pulse, of width τ_{bb} s, and filtering the channel response with a bandpass filter, with bandwidth $\frac{2}{\tau_{bb}}$ Hz. This filtered response is measured and saved using an oscilloscope, and the saved measurement is known to be the convolution of the CIR and the transmitted pulse. This measurement is as close as one can come to directly measuring the impulse response of a channel.

The resolution of the system is equal to the width of the transmitted pulse; in other words multipath components received within time periods shorter than this transmitted pulse will not be detected. Due to this constraint, narrow pulses are preferred for a higher temporal resolution (i.e., a wide bandwidth system is needed). A major problem with this approach is that the system relies heavily on the oscilloscope to trigger with the arrival of the first signal (typically the LOS component of most power). This can cause the system to sometimes not trigger properly when operating in highly multipath environments, or environments where the LOS component is blocked (such as the environments relevant to this work [5], [3]). An advantage of the system is that it can be constructed using off the shelf hardware present in most wireless communications laboratories, assuming the desired bandwidth requirements are met. There exist other methods of time domain channel sounding, such as spread spectrum sliding correlator presented in [15], and orthogonal frequency division multiplexing (OFDM) based channel estimation methods [16], but the rest of this section will focus on channel characterization in the frequency domain.

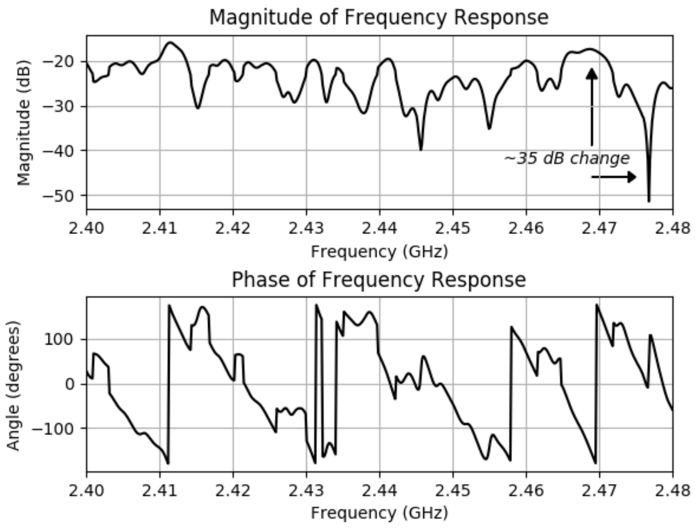


Figure 2.5: An example S_{21} plot for a measurement in a highly multipath wireless channel over the 2.4 GHz ISM Band (2.40 GHz - 2.48 GHz). Note the sharp transition in signal strength at around 2.47 GHz labeled on the figure.

2.3.2 Frequency Domain Channel Measurements

Channel measurement in the frequency domain typically involves transmitting tones at incremental frequencies and measuring the magnitude and phase of these tones at the receiver. The goal of frequency domain channel characterization is to obtain the frequency response of the channel, and is typically measured using a Vector Network Analyzer (VNA). The measurement produced by the VNA is called a S_{21} measurement, and this measurement gives information on the path loss over the channel over frequency.

An example S_{21} plot is displayed in Fig. 2.5, magnitude displayed on top and phase on the bottom. The data presented in Fig. 2.5 was captured using a VNA within a test chamber at the University of Vermont, referred to as the Compact Re-configurable Channel Emulator (CRCE). This chamber is designed to mimic the channel conditions present in emerging IoT applications (e.g., a metallic air frame, a

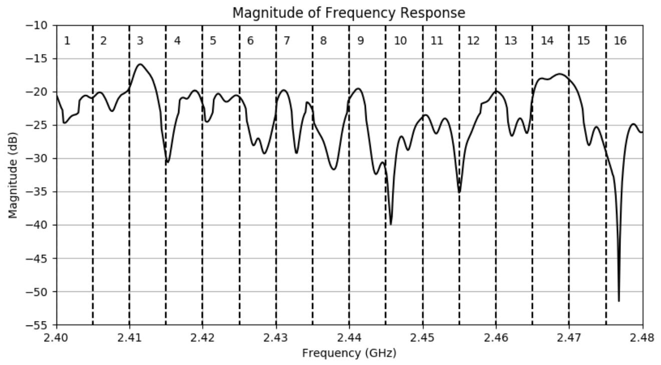


Figure 2.6: The magnitude of the S_{21} plot from Fig. 2.5 with the IEEE 802.15.4 channels superimposed for clarification. The channel numbers are the numbers along the top of the plot. Channel 14 exhibits the most benign fading and Channel 16 the most severe.

factory floor, machine-to-machine communications, etc.). The chamber was originally presented in [7] and was recently updated in [8]; it will be used throughout this thesis as a testing environment for multipath channel conditions.

It can be seen from the magnitude plot in Fig. 2.5 that there is a drop in signal strength of $\sim$ 35 dB between $\sim$ 2.470 GHz and $\sim$ 2.477 GHz. That is a significant change in signal strength ($\geq$ three orders of magnitude) over a relatively small change in frequency ($\sim$ 7 MHz). Changes in signal strength that are frequency dependent are referred to as frequency selective fading [17], and significant changes such as this would be considered *severe* frequency selective fading.

This information would not be evident from the time domain channel impulse response, and these insights into the performance of individual channels can be very useful in determining the reliability of a wireless system in an environment. In other words, while the time domain channel sounding methods give a “tangible” channel model (i.e., information on the times at which the multipath components reach the receiver), the frequency domain channel measurements can give intuition into system performance and constraints.

As an example, the IEEE 802.15.4 wireless standard [18] is a common wireless standard in low data rate, low power wireless sensor networks, and is what ZigBee [19] and other similar protocols are based on. This standard utilizes the 2.4 GHz Industrial Scientific and Medical (ISM) band, which is 80 MHz wide and extends from 2.40 GHz to 2.48 GHz. Inside of this 80 MHz band, 802.15.4 assigns 16 channels of operation of width 5 MHz. These channels are superimposed onto the magnitude response from Fig. 2.5 in Fig. 2.6. From this figure it can seen that this frequency selective fading would cause channel 16 to be virtually inoperable while channel 14, and possibly even channel 15 would allow for reasonable data transfer. Based on this measurement, decisions can easily be made about which channels would be better to operate over in this environment. This is but one example of why the frequency domain response of a wireless channel can be useful.

A limitation of these measurements are that the costly VNA is typically a coherent system, meaning that the transmitting and receiving antenna need to be connected to the same device (the VNA). This means that characterization over large channels using this method is difficult, and requires the use of very long cables. It would therefore be desirable to have a *distributed*, *portable*, and *low cost* measurement system that would provide reliable frequency response measurements, which is what this work aims to do.

2.4 Conclusion

In this chapter, the basic theory underlying signal propagation was presented. The differences between large and small scale fading, as well as their respective mathematical models were examined and discussed in detail. Lastly, existing methods of empirically collecting channel data to be used to characterize wireless channels using the aforementioned mathematical models were explored. Both time and frequency domain methods for empirical channel data collection were presented, as well as the potential benefits and drawbacks of each method. As mentioned prior, the work conducted in the Wireless Communications Lab at the University of Vermont is mostly interested in small-scale propagation effects and typically involves frequency domain channel measurements (i.e. using a VNA). This thesis work aims to develop a system which can measure the scalar frequency response of a channel, in a distributed manner. Ideally this system would be low cost and portable. A recent topic of research and of interest to hobbyist is the software-defined radio (SDR), which this thesis utilizes to perform the frequency domain measurements, and which will be introduced in the next chapter.

Chapter 3

Software-Defined Radio

A software-defined radio (SDR) is a radio which implements all signal processing tasks involved in wireless communication (e.g., modulation, demodulation, pulse shaping, filtering, etc.) in digital signal processing (DSP) as opposed to analog circuitry. The major benefit of implementing a radio in such a way is the ease of reconfigurability of software versus hardware; this opens the door for radios which can quickly reconfigure in real-time as well as for rapid prototyping in development. In this thesis, a novel wireless channel measurement system is developed which uses SDR. This chapter will first examine the history and motivation of software-defined radios, followed by discussing the specific hardware and software used in this work. The last section will introduce prior work done using SDR for wireless channel measurements.

3.1 History and Motivation

The origins of software-defined radio come from the desire for a reconfigurable radio. Such a radio would in theory be able to perform the tasks of multiple hardware radios in one device, and this was originally desired for military applications. According to [20], the term "software radio" was first used in a company newsletter issued internally by Raytheon (then E-Systems) referring to a prototype digital receiver which implemented adaptive filtering using an array of processors [21]. In the early 1990's, the Defense Advanced Research Projects Agency (DARPA) also led a program called SpeakEASY which aimed to "use programmable processing to emulate more than 15 existing military radios" [22]. While this DARPA project did not explicitly mention the term software defined radio, it is clear that their vision was of a radio based in software.

The first time the term is used in academic literature is in Joseph Mitola's paper [23] published in IEEE in April 1993. This paper defined the software radio in its present form as follows:

A software radio is a set of Digital Signal Processing (DSP) primitives, a metalevel system for combining the primitives into communications systems functions (transmitter, channel model, receiver ...) and a set of target processors on which the software radio is hosted for real-time communications. - J. Mitola [23]

From Mitola's definition, it can seen that the software radio encompasses both the software (the DSP primitives) and the hardware (the target processors) needed for a functional system. Because the E-Systems software radio was really only a receiver Mitola is credited with defining the software radio in how it is viewed today, as a fully functional radio defined in software. He is also responsible for bridging the interest gap into the field of traditional academia (and not solely military applications).

Along the way, the term software radio evolved into software-defined radio (SDR) as it is known today. These early examples of research into software-defined radio (namely the Speakeasy project) are not always considered successful as the technology at the time sometimes fell short of the necessary hardware requirements (e.g., most notably speed and timing limitations of the analog-to-digital and digital-to-analog converters). But as the hardware technology has improved over the last two decades, this has caused a resurgence in interest into research related to software-defined radios. The next section will take a look at this hardware more closely.

3.2 Hardware

The ideal software-defined radio would consist of only a select few hardware components. Namely an antenna and analog-to-digital converter (ADC) on the receive side, and an antenna and digital-to-analog converter (DAC) on the transmit side. The conversion between the analog and digital signal domains is crucial to the operation of SDRs and will be examined in the following subsection.

3.2.1 Digital Processing of Analog Signals

Often times, it is desirable to approximate an analog signal by a digital signal so that one can utilize digital signal processing (DSP) techniques, which can have benefits over analog processing in some scenarios (e.g., higher order filters can be simpler to implement, easier to reconfigure filter, device tolerances not as much of an issue, etc.).

The block diagram of a typical analog-to-digital converter is displayed in Fig. 3.1. The analog signal $x_a(t)$ enters the system, is lowpass filtered by an anti-aliasing filter $F_a(\Omega)$, is sampled by an ideal sampler of period T_s, and lastly is amplitude quantized

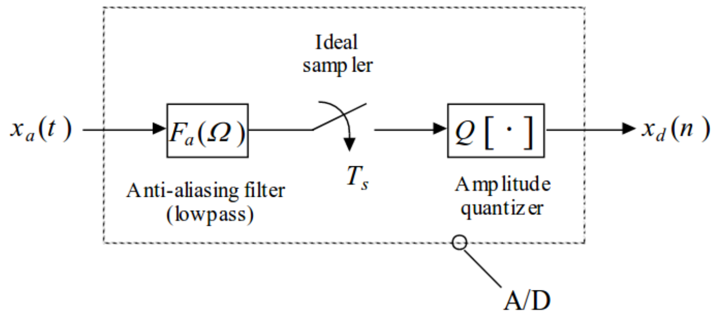


Figure 3.1: Block diagram of an analog-to-digital converter [25].

$Q[\cdot]$ before the new digital signal $x_d(n)$ is output from the system. Sampling is the core process underlying analog-to-digital conversion, and information is inherently lost in the process. In general to sufficiently sample a signal, the sampling frequency Ω_s must be higher than the Nyquist rate Ω_N, as defined below [24]:

$$\Omega_N > 2\Omega_b \tag{3.1}$$

Where Ω_b is the bandwidth of the signal to be sampled. Note, $\Omega_s = 2\pi f_s = T_s$ where Ω_s is in $\frac{radians}{second}$, f_s is in Hz, and T_s is in seconds. When the sampling rate is not high enough (i.e., $\Omega_s \leq \Omega_N$), the signal will be under sampled and aliasing will occur. Aliasing (also called spectral folding [24]) will cause higher frequency content to appear at lower frequencies in the sampled signal, and will severely distort the signal. The anti-aliasing filter $F_a(\Omega)$ aims to prevent this from occurring by filtering out any unnecessary higher frequency components ($f > \frac{f_s}{2}$) of the signal. After the signal is sampled, it is quantized to discrete amplitude values so that it can be represented in binary form.

The dual of analog-to-digital conversion is digital-to-analog conversion, and a block diagram of this process is displayed in Fig. 3.2. In this process, the digital signal $y_d(n)$ enters the system, goes through a digital compensating filter $\Gamma_d(e^{jw})$, is

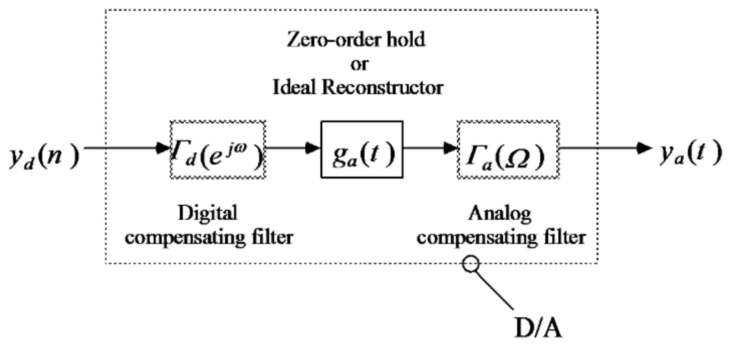


Figure 3.2: Block diagram of a digital-to-analog converter [25].

reconstructed in $g_a(t)$, and lastly goes through an analog compensating filter $\Gamma_a(\Omega)$ before the analog signal $y_a(t)$ is output from the system. The core process underlying digital-to-analog conversion is the reconstruction process, which aims to interpolate the signal between samples. The ideal reconstructor is displayed below [24]:

$$g_a(t) = \frac{sin(\frac{\pi t}{T_s})}{\frac{\pi t}{T_s}} \iff G_a(\Omega) = \begin{cases} T_s & |\Omega| \leq \frac{\pi}{T_s} \\ 0 & \text{else} \end{cases} \tag{3.2}$$

The ideal reconstructor is an ideal low-pass filter, whose pass-band is equal to the sampling frequency. In the time domain this becomes a series of sinc functions centered around each sample. Mathematically this reconstruction works in the ideal case of a perfectly band-limited signal, but in reality no signals are perfectly band-limited and ideal low-pass filters are not realizable. For this reason, a more common method of reconstruction is the zero-order hold (ZOH) reconstructor, displayed below [24]:

$$g_a(t) = \begin{cases} 1 & 0 \leq t < T_s \\ 0 & \text{else} \end{cases} \iff G_a(\Omega) = \frac{sin(\frac{\Omega T_s}{2})}{\frac{\Omega}{2}} * e^{-j\Omega\frac{T_s}{2}} \tag{3.3}$$

The ZOH reconstructor is essentially the opposite of the ideal reconstructor, with a rect function in the time domain, and sinc function in frequency. In this case, the reconstructor *holds* each sampled value until the subsequent sample, keeping $y_d(n)$ at a constant value between samples. These sharp transitions between samples introduce high frequency content to the signal which must be compensated for. This compensation can be done in the digital domain ($\Gamma_d(e^{jw})$), the analog domain ($\Gamma_a(\Omega)$), or some combination of the two.

Unlike the ideal reconstructor, this method works reasonably well in practice. There exist other interpolation methods (e.g., first-order hold, second-order hold, etc.) but for the sake of this thesis these are enough to understand the digital-to-analog conversion theory.

3.2.2 Hardware Implementation in SDR

In order for modern day SDRs to be relatively low-cost, accurate, and still operate at RF frequencies, an analog intermediate frequency (IF) signal is typically used in between the digital and RF domains. The method of converting an RF signal to IF for processing is referred to as a *superheterodyne receiver*. At the receiver the analog RF signal is down-converted to an IF before ADC conversion, and likewise at the transmitter the digital signal is converted to an IF before being up-converted to RF for transmission.

A flow graph for a typical superheterodyne analog home radio system is displayed in Fig. 3.3. The wireless RF signal enters the device through the antenna, is amplified by the RF amplifier and converted to a lower frequency (IF) by the analog mixer. The local oscillator frequency is typically set by the radio tuner to select which radio

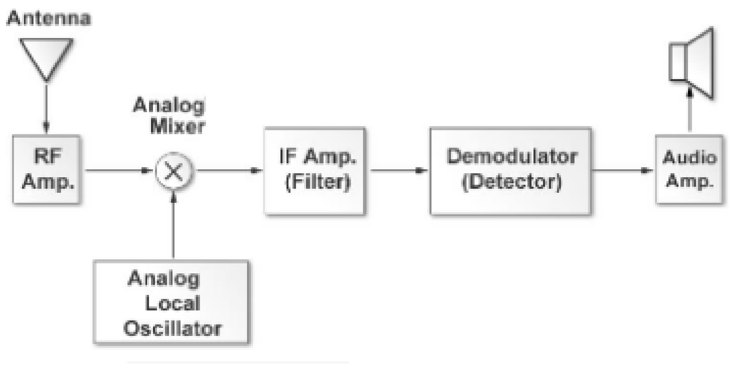


Figure 3.3: Flow graph of an analog receiver [26].

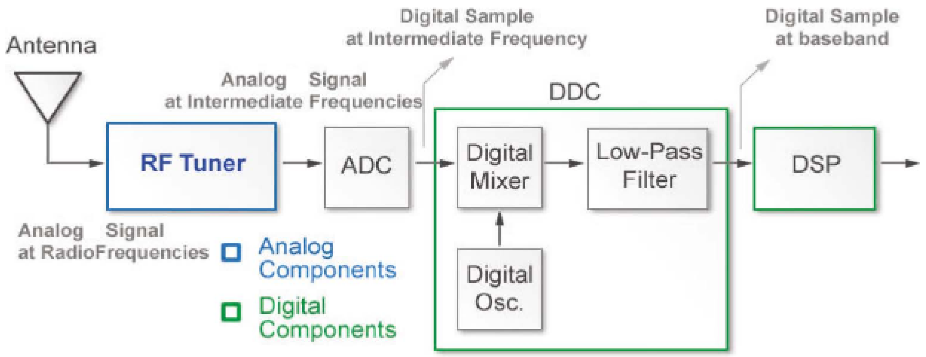

Antenna
Analog Signal at RadioFrequencies
RF Tuner
Analog Signal at Intermediate Frequencies
ADC
Digital Sample at Intermediate Frequency
DDC
Digital Mixer
Digital Osc.
Low-Pass Filter
Digital Sample at baseband
DSP
Analog Components
Digital Components

Figure 3.4: Flow graph of a typical SDR receiver [26].

station to tune into. After down-conversion, the IF signal is amplified and then demodulated to recover the baseband signal. In the radio in Fig. 3.3, the baseband signal is output through a speaker.

The software-defined alternative to the radio in Fig. 3.3 is displayed in Fig. 3.4. In Fig. 3.4, the analog mixer, RF amplifier, and IF amplifier are contained within the RF tuner. In the SDR, the signal is converted from analog to digital at this point using an ADC. This IF digital signal is converted to baseband digitally using a Digital Down-Converter (DDC). The DSP (digital signal processing) block represents any processing that is done to the baseband signal (i.e., demodulation, decoding, etc.)

The typical transmitting SDR has a similar structure, which is displayed in Fig.

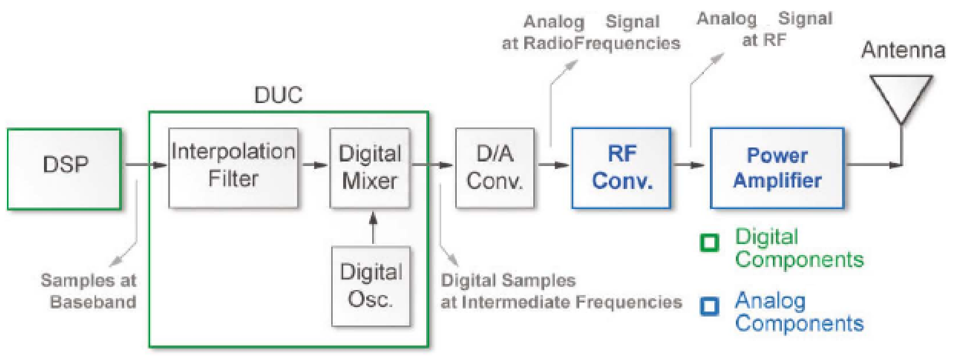


Figure 3.5: Flow graph of a typical SDR transmitter [26].

3.5, also taken from [26]. In this case, the signal is first processed digitally before being sent to a digital-up-converter (DUC) to convert to IF. An interpolation filter is needed to increase the sample rate of the signal before the digital mixing. After the DUC, the digital signal is now at a higher frequency (IF). The digital IF signal is then converted to analog IF through a DAC, and then up-converted to RF. Due to high-loss at RF frequencies in wireless environments, the RF signal is amplified before transmission.

3.2.3 SDR Hardware Examples

There are many commercially available software-defined radios with a wide range of performance specs and price points, and a few examples are displayed below in Table 3.1. While a hobbyist may wish to use the RTL2832 USB Dongle to listen to FM radio stations, a research group may need the performance of the USRP X310 to test novel communication protocols. When choosing the best SDR to use in this work there were a few things to consider.

As mentioned in a prior section, this work aims to develop a novel channel measurement system using software-defined radio. The small-scale channel effects mentioned in Section II are especially applicable in wireless sensor networks, which tend

Radio	Price	Frequency Range	Bandwidth	Transceiver?	ADC Resolution
RTL2832 USB Dongle	\$22.50	24 MHz - 1766 MHz	3.2 MHz	No (only RX)	8-bits
HackRF One	\$299.95	1 MHz - 6 GHz	20 MHz	half-duplex	8-bits
BladeRF x40	\$420.00	300 MHz - 3.8 GHz	28 MHz	full-duplex	16-bits
USRP X310	\$5,290.00	DC (0 Hz) - 6 GHz	160 MHz	full-duplex	16-bits

Table 3.1: Various software-defined radio hardware packages available for purchase and their respective performance specs.

to operate in the 2.4 GHz Industrial, Scientific, and Medical (ISM) band. This band extends from 2.40 GHz to 2.48 GHz and is the frequency range that this system will operate over. So, a SDR which can operate at these frequencies is necessary for this work.

Ideally the system will also be low-cost, but a trade-off among these devices is that lower-cost devices tend to have smaller operational bandwidths. Most prior work implementing channel measurement systems using SDR (to be discussed further Section 3.4) use radios whose operational bandwidth exceeds the bandwidth over which they would like to measure. One of the contributions this work makes is using a radio with a smaller operational bandwidth than the band to be measured over, and subsequent synchronization algorithms. Considering these factors on frequency range, operational bandwidth, and cost, the HackRF and BladeRF radios were chosen as the radios to be used in this work. Now that the hardware behind SDRs has been presented, the accompanying software will be examined next.

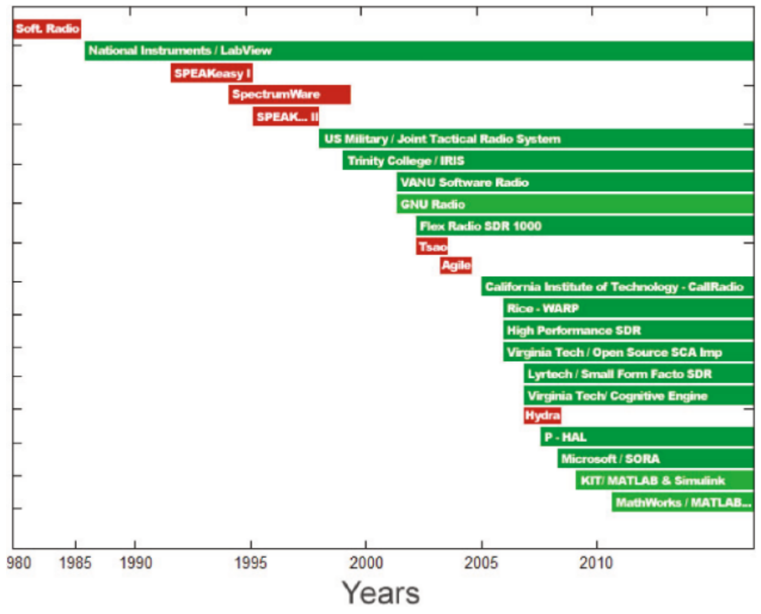
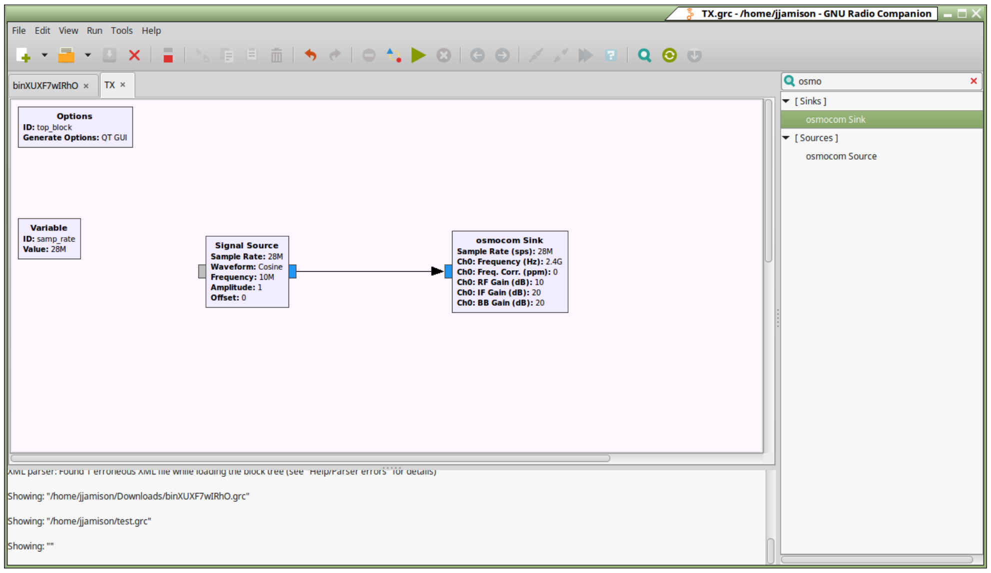


Figure 3.6: The many SDR software frameworks and the years they were active [26].

3.3 GNURadio

Many software frameworks have been developed for use with software-defined radios since their inception in the 1980's, as shown in Fig. 3.6. GNURadio is one of the most popular frameworks used in present time, as it is an open source software with an extensive well documented API [31]. GNURadio allows program creation in either Python or C++, and includes a flow-graph based SDR workbench called GNURadio Companion (GRC). GNURadio also supports most popular SDRs on the market today, including the HackRF and BladeRF. GNURadio was the software chosen for this work due to its high degree of customizability, and its open-source nature.

Fig. 3.7 displays a screen shot of a basic program created in GNURadio Companion which will transmit a tone at 2.401 GHz through the SDR connected to the computer running the program. The *osmocom sink* block is the block used to trans-



Options
ID: top_block
Generate Options: QT GUI

Variable
ID: samp_rate
Value: 28M

Signal Source
Sample Rate: 28M
Waveform: Cosine
Frequency: 10M
Amplitude: 1
Offset: 0

osmocom Sink
Sample Rate (sps): 28M
Ch0: Frequency (Hz): 2.4G
Ch0: Freq. Corr. (ppm): 0
Ch0: RF Gain (dB): 10
Ch0: IF Gain (dB): 20
Ch0: BB Gain (dB): 20

```
XML parser: Found 1 erroneous XML file while loading the block tree (see "Help/Parser errors" for details)

Showing: "/home/jjamison/Downloads/binXUXF7wIRhO.grc"

Showing: "/home/jjamison/test.grc"

Showing: ""
```

Figure 3.7: A basic transmitting program in GNURadio Companion. A signal source block and osmocom sink block are used in this program.

mit through the HackRF and BladeRF radios. The *signal source* block generates a cosine wave of 10 MHz in its displayed configuration, and the *osmocom sink* block transmits the signal through the affiliated radio with a center frequency of 2.4 GHz. In GNURadio, when a 10 MHz tone is transmitted with a center frequency of 2.4 GHz, it is up-converted to 2.401 GHz. The *RF Gain*, *IF Gain*, and *BB Gain* parameters set the various gain settings for the radios.

There are two ways to develop custom functionality in GNURadio. One way is to create custom blocks in either Python or C++ to be used in GRC. This can be helpful when writing a custom filter, demodulator, or some other kind of DSP functionality that will run the same way on incoming data throughout the duration of the program. A limitation of GRC is that there is no straight forward way to implement loops, either graphically or in custom blocks. This is due to the nature of the protocol the GNURadio developers use for streaming data from one block to the next.

The other way to develop custom functionality is to write Python scripts using the GNURadio functional libraries. In this manner, one can still use the included function blocks in GNURadio (as programming objects) without the limitations of the GRC GUI environment (e.g., the lack of loops). Whenever a block diagram is run in GRC a Python script is automatically generated. Therefore the most straight forward approach to writing custom scripts using these libraries is to create a basic block diagram in GRC, run the program to generate the Python script, and then edit this script to add in whatever custom functionality is needed. This is the method used in this work.

3.4 SDR Based Channel Characterization

Developing systems for wireless channel measurements using software-defined radio is a topic of recent research, and this section will detail the existing work on this topic. The advantage of measuring wireless channels with software-defined radios is evident both in price and portability. For the most part, even the more expensive SDRs are significantly cheaper than the devices needed for traditional channel sounding methods. SDR based systems also allow for easy transfer of technology. For example, the algorithms which enable a pair of SDRs to measure a wireless channel can be saved and loaded onto two other SDRs which now also become a measurement system.

The majority of prior work on this subject has focused on time domain measurements (i.e., measurements to obtain the channel impulse response (CIR) or power delay profile (PDP)). In [32] a channel sounder was created in which one software defined radio transmitted a chirp signal, and another put the received signal through a matched filter. Using the output from the matched filter the algorithm was able to estimate the CIR. This system focused on time-domain information and had a measurement bandwidth of 25 MHz, not very useful when considering devices operating in the 2.4 GHz ISM band.

A second time-domain sounding method proposed in the literature is to transmit orthogonal frequency division multiplexing (OFDM) symbols with periodic pilot symbols and use OFDM channel estimation techniques [33, 34, 35]. Though this technique works very well for SDRs with large instantaneous bandwidths, these techniques are harder to apply to lower bandwidth devices. These measurements are also focused on time domain information, although it is worth noting that if the measurement system is coherent, as in [35], the frequency response of the channel can be obtained through a Fourier transform.

For the cases relevant to this work, the frequency response over a bandwidth wider than the bandwidth of the SDRs being used is desired, and this is not the case in any of aforementioned systems. The system in [36] comes the closest to achieving the desired measurement. This system uses two BladeRF SDRs to obtain frequency response measurements over 43.5 MHz bandwidth by utilizing what the author calls over-lapping tone and power detection algorithm. Essentially a collection of eight tones are transmitted over a 2.4 MHz band (yielding 300 kHz resolution). After a period of time the transmitting radio increments its center frequency by 2.4 MHz Hz, and when the receiving radio detects that the received power has dropped (because the tones have moved) it also moves its center frequency. To the author's knowledge,

	Time or Frequency	System Bandwidth	Measurement Bandwidth	Center Frequency	Distributed?
J. Li et. al [35]	Both	20 MHz	200 MHz	4.9 GHz - 5.9 GHz	No
H. Boeglen et. al [33]	Both	160 MHz	50 MHz	2.3 GHz and 5.8 GHz	No
T. Srisooksai et. al [36]	Both	2.4 MHz	43.5 MHz	2.45 GHz	No
N. Hosseini et. al [32]	Time	100 MHz	25 MHz	2.42 GHz	No
This System	*Frequency*	*10 MHz*	*80 MHz*	*2.44 GHz*	*Yes*

Table 3.2: A summary of the significant prior works listed in chronological order.

this is the only instance in the literature of a channel measurement system based in software-defined radios, that directly measures the frequency response. This thesis uses the work presented in [36] as inspiration and significantly extends it. Table 3.2 provides a chronological summary of the prior works discussed along with the specifications of the system proposed by this thesis.

3.5 Conclusion

In this chapter, software-defined radio was introduced. First the history and motivation behind radios based in software was discussed, before getting into the hardware and software behind their operation. Lastly, existing channel measurement systems based in SDR were examined. The next chapter will detail the operation of the system being proposed by this thesis, and its improvements from the previously mentioned systems for measurements in static highly multipath environments.

Chapter 4

Proposed Measurement System

As stated in prior chapters, this thesis aims to develop a system which can measure the frequency response of a wireless channel using software-defined radio. The developed system is able to measure the scalar frequency response of a 80 MHz channel to within 0.018 % σ of measurements taken with a VNA in highly multipath environments were the signal strength can vary $\sim$ 40 dB over the 80 MHz bandwidth of interest and it does this all in a distributed manner (i.e., the transmitting and receiving "ends" of the system are not physically connected in any way). Furthermore, the system is capable of scaling to measuring multiple links simultaneously. The operation of the system as well as some verifying proof-of-concept measurements are presented in this chapter.

4.1 Measurement Methodology

For purposes relevant to this work, the scalar frequency response is the information desired about the channel, and as discussed in Section 2.3.2, frequency response measurements are typically obtained using a Vector Network Analyzer (VNA). To make a measurement, the VNA transmits tones at incremental frequencies and records the magnitude and phase of the received tones, before interpolating the areas between the measurements to obtain the frequency response of the channel. In other words, a VNA samples the frequency response at various frequencies throughout a band. This method of sampling the frequency response of the channel is uses in the proposed system as well. Since a VNA is a coherent measurement system, the system is capable of measuring both the magnitude and phase of the signal (i.e. it takes a *vector* measurement). As mentioned in Section 2.3.2, oftentimes in highly multipath environments the magnitude of the frequency response is more useful than the phase. Thus for our purposes we will focus on a *scalar* measurement system which measures the magnitude of the frequency response.

4.1.1 Measurements Using "Chirped" Signals

In order to transmit tones at incremental frequencies, a Linearly Frequency Modulated (LFM) signal is used. The LFM signal, also called a "chirp" signal, is a sinusoidal tone whose instantaneous frequency ($f(t)$) varies linearly with time.

$$\begin{aligned} x(t) &= Acos(2\pi f(t)t + \phi) \\ f(t) &= \frac{k}{2}t + f_0 \end{aligned} \tag{4.1}$$

The equation of a LFM signal, whose frequency sweeps from f_0 to f_1 at a rate of $k = \frac{f_1 - f_0}{T}$ is displayed above in Eq. 4.1. Similar signals are commonly used in radar and sonar applications were information is needed at incremental frequencies in order to identify target distances. By using this type of signal, the frequency response can be sampled at incremental frequencies. Since the operational bandwidth of the radios being used is 20 MHz (28 MHz for BladeRF), it is evident that the center frequency of the SDR will need to be stepped intermittently throughout an 80 MHz wide measurement. In theory if repeatable measurements can be taken over the entire 20 MHz, the center frequency would need to be stepped four times.

To test if measurements taken over 20 MHz of bandwidth were repeatable, an experiment was run where a chirp signal was transmitted from a HackRF, through a 20 dB attenuator, and fed directly (via cabling) to the receiving port of a BladeRF. In this way, the "channel" under test was completely hardwired and constant. The chirp signal varied from 2.4 GHz, to 2.42 GHz and stepped in 78.125 kHz increments. The increment size was chosen such that there were 256 steps over the 20 MHz band, so that at the receiver a 256 point FFT could be used to measure the magnitude of each "step" of the chirp, with each FFT bin corresponding to a unique frequency of the chirp. Technically this is not a true LFM signal, since the frequency is not swept from one point to another continuously, but is rather stepped from one increment to the next discretely.

The transmitting and receiving programs were run on separate computers. One computer ran a script which generated the tones and transmitted these tones from the HackRF, and a second computer ran a script which would sample the received signal at the BladeRF and save the raw samples to a file. The center frequency of both the transmitting and receiving SDRs was kept constant at 2.41 GHz. A third script was written to compute the frequency response from this sampled data, using a 256-point FFT in post processing (it was found that computing the FFT in real-time

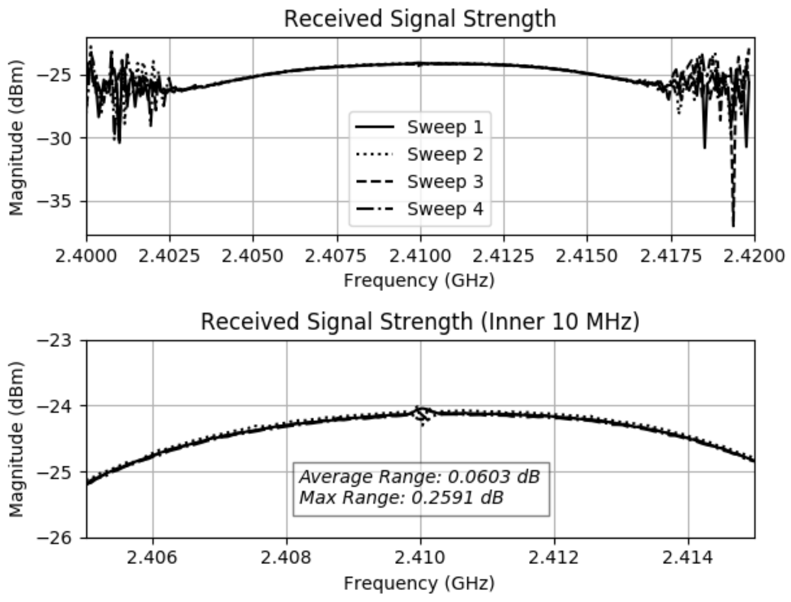


Figure 4.1: "Through" measurement testing the repeatability of frequency response measurements using a chirp signal, and a BladeRF / HackRF combination. Full 20 MHz possible bandwidth (top), the inner 10 MHz (bottom).

led to CPU overflow, resulting in dropped samples).

This measurement was conducted four consecutive times and the results of each sweep are provided in Fig. 4.1. The top plot displays the measured frequency response over the full measurement (20 MHz) and it can be seen that as the measurements extend further from the center frequency, they become much less repeatable. The bottom plot shows the inner 10 MHz (from 2.405 GHz - 2.415 GHz) of the measurement, and it can be seen that this inner range is much more repeatable. Between the four sweeps, the maximum deviation between sweeps over this 10 MHz is 0.26 dB, and the average deviation is a mere 0.06 dB. Since these measurements are repeatable over this bandwidth, it would be possible to calibrate out the slope of this "through" measurement over the inner 10 MHz of the measurement. In short, while the hardware is capable of 20 MHz of measurable bandwidth we find using only 10 MHz of bandwidth is significantly more repeatable thus will be used as the limit for

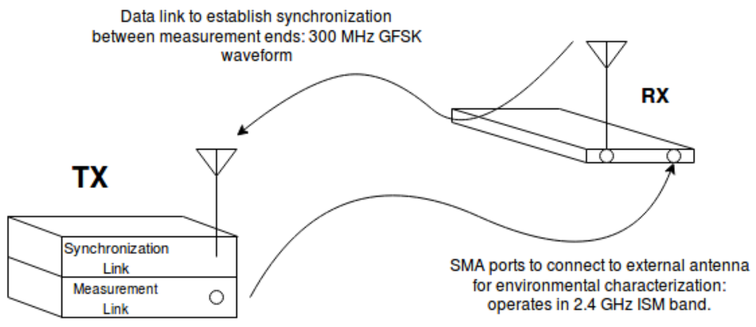

Data link to establish synchronization between measurement ends: 300 MHz GFSK waveform
RX
TX
Synchronization Link
Measurement Link
SMA ports to connect to external antenna for environmental characterization: operates in 2.4 GHz ISM band.

Figure 4.2: A schematic of the one-link measurement system. On the transmitting (TX) end two HackRFs are stacked on top of one another, one for the synchronization link and another for the measurement link. One the receiving (RX) end, one BladeRF is used for both links.

any *single* measurement in our system.

4.1.2 Establishing Synchronization

With 10 MHz of repeatable measurement bandwidth, the system will need to switch its center frequency eight times in order to measure over the desired 80 MHz. Due to the distributed nature of the system, some sort of communication link needs to exist between the two measurement ends so that messages can be sent to synchronize these center frequency steps. Since the BladeRF is a full-duplex transceiver with separate transmit and receive ports, the radio is capable of sending messages while simultaneously taking measurements. As for the HackRF, only one port exists for both transmitting and receiving, therefore two HackRF need to be used to achieve full duplex communications. For these reasons one BladeRF is used at the receiving end of the measurement system, and two HackRF are used at the transmitting end of the measurement system, this is displayed in Fig. 4.2. The synchronization link is established using an antenna directly connected to the devices, while the measurement link is established through external antenna connected through the SMA ports at each measurement end.

For the synchronization link, Gaussian frequency shift keying (GFSK) was used as the data modulation scheme. Traditional frequency shift keying (FSK) modulation involves transmitting a signal which changes frequency based on the data to be transmitted [17]. These quick changes in frequency can cause spectral content to appear outside of the intended transmission frequencies, which could cause issues when this is occurring in a measurement system. GFSK modulation involves passing the digital data through a Gaussian filter before being FSK modulated [37] to reduce the magnitude of this extraneous spectral content. GNURadio has built-in functions for GFSK modulation and demodulation which were used to accomplish this.

This data link was established at 300 MHz so as not to interfere with the measurements taking place at higher frequencies and because lower frequencies tend to be less affected by the multipath environments this system aims to characterize. As longer wavelengths are less susceptible to scattering effects [9], lower frequencies should be effected less by the multipath allowing the system to have a reliable synchronization link. The packet structure used for this data link is displayed below:

Preamble:	Payload Length	Payload	XOR Checksum
0xAA	8 - bits	16 - bits	key: 0xFAFA

For this work, the payload is always 16-bits long (thus payload length bits are always 0x10) and contains the serial address of the radio transmitting the message. This is especially important when characterizing over multiple links, so the transmitter can differentiate between messages received from different radios. The preamble and XOR checksum key are also *always* the values displayed above. Python programs were written to control the radios while taking measurements over 10 MHz of band-

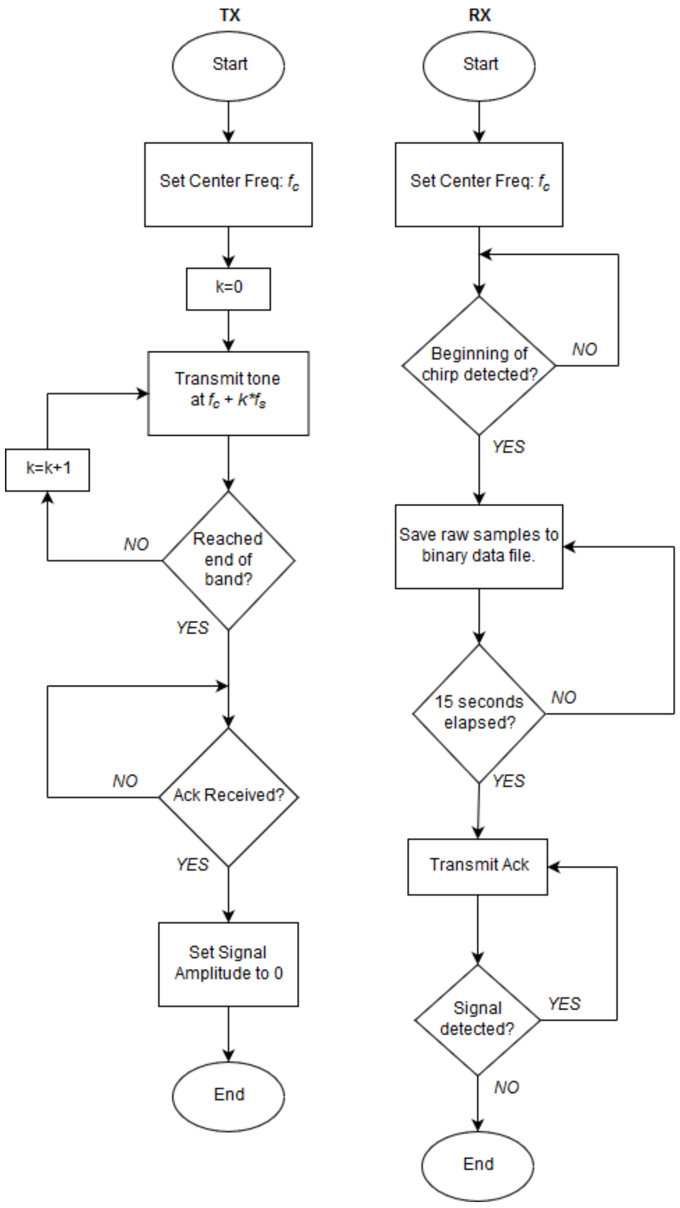


Figure 4.3: Flow charts displaying the program flow of the transmitting and receiving Python programs called by the shell scripts in Fig. 4.4. f_s is the frequency increment between tones in the chirp (78.125 kHz), and the end of the band is determined when k = 128 (for 128 78.125 kHz steps over the 10 MHz band). Source code available in Appendix A.1.2 (transmitting) and A.2.2 (receiving).

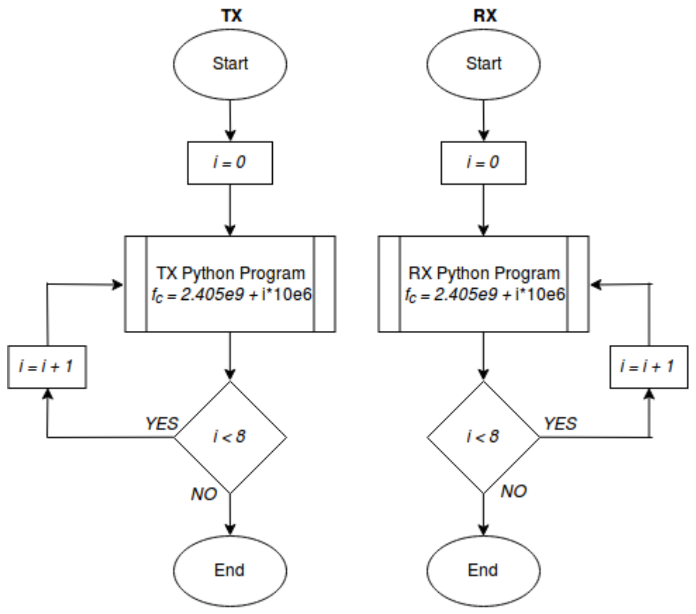


Figure 4.4: Flow charts representing the program flow of the shell scripts used to call the transmitting and receiving Python programs in Fig. 4.3. Source code available in Appendix A.1.1 (transmitting) and A.2.1 (receiving).

width, while shell scripts were written to call these programs with intermittent center frequencies. This approach allowed for the entire 80 MHz to be swept over.

The program flow of these shell scripts are displayed in Fig. 4.4, while the program flow of the Python programs are displayed in Fig. 4.3. The transmitting and receiving programs correspond to completely separate ends of the measurement system (i.e. control different radios and are run from different computers in a distributed fashion). From Fig. 4.4 it is seen that the shell script calls the Python program at eight incremental center frequencies (2.405 GHz, 2.415 GHz, 2.425 GHz, ... , 2.475 GHz). At each step, the transmitting Python program transmits a chirp, and the receiving Python program saves the sampled versions of the signals it receives to a file, until it determines that the chirp has ended. This point is determined by detecting the first point that a signal is present, and then waiting the approximate time it takes for the chirp to run (15 seconds). After this 15 seconds, the program begins sending ac-

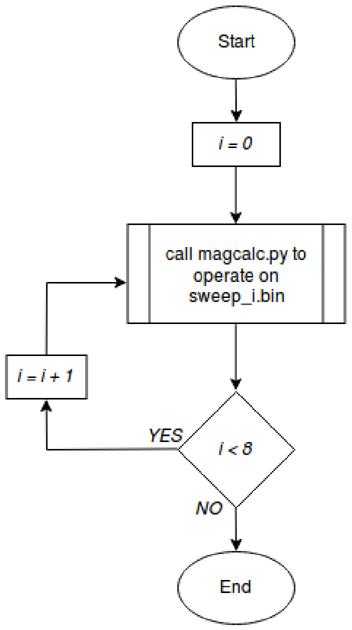


Figure 4.5: Flow chart displaying the program flow of the shell script which calculates the channel frequency response from the saved .bin files. Source code available in Appendix A.3.1.

knowledgement messages to the transmitter indicating the measurement is complete. Once the transmitter receives one of these messages, it sets the signal amplitude to zero and exits to the shell script (where the center frequency will be incremented). Once the receiving program detects that the signal amplitude has been set to zero, it also returns to the shell script, for its center frequency to be incremented as well.

4.1.3 Frequency Response Calculation

When the full shell script has completed running, there will be eight .bin files saved to the PC of raw IQ sampled data, one corresponding to each frequency step. The frequency response needs to be calculated from this data in post-processing and a Python script was written for this purpose. In a similar manner to the programs presented in Fig. 4.4 and Fig. 4.3, a Python script is written which computes the frequency response over 10 MHz from a single .bin file, and a shell script is written

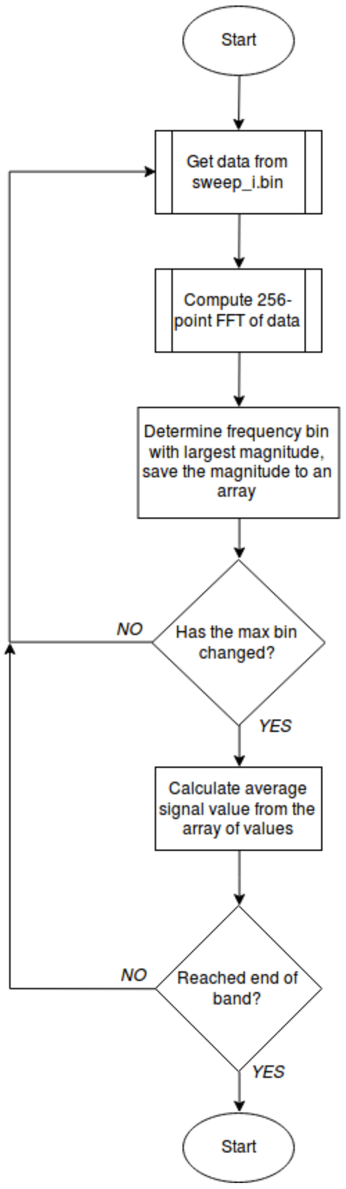

Start
Get data from sweep_i.bin
Compute 256-point FFT of data
Determine frequency bin with largest magnitude, save the magnitude to an array
Has the max bin changed?
NO
YES
Calculate average signal value from the array of values
Reached end of band?
NO
YES
Start

Figure 4.6: Flow charts displaying the program flow of the Python program called by the shell script in Fig. 4.5. Source code available in Appendix A.3.2.

to call this Python script eight times for each .bin file and successfully computing the scalar frequency response over the entire 80 MHz bandwidth.

The program flow of the shell script used to calculate the frequency response is displayed in Fig. 4.5. This script successively calls the Python program displayed in Fig. 4.6 eight times, specifying which file to operate on each call. Each run the Python program opens a file and attempts to track the chirp signal as it moves across the band using an FFT. The FFT used is a 256-point FFT and is calculated every 256 data points. Since the wireless signal is sampled every 50 ns ($f_s = 20$ MHz) a 256-point FFT run in real-time is calculated every 12.8 μs. Trying to do this calculation in real-time leads to CPU overflows, so all calculations are done in post-processing. A 256-point FFT is used so that each step of the chirped signal corresponds to a bin in the FFT, and so that the temporal resolution is sufficiently small enough that there will be few cases were an FFT is calculated when the chirped signal is in the process of stepping between frequencies (i.e., less of a chance of computing the FFT at times when two signal steps are present).

Every time the Python program in Fig. 4.6 is called, it reads the binary file step_i.bin (where i corresponds to the step number), and saves the frequency response over this 10 MHz increment to a text file step_i.txt. The text file is significantly smaller than the raw sampled data, as the text file will contain 128 data points for the 10 MHz band, while the raw data file contains *many* more data points (e.g. sampling at 20 MHz for 20 seconds, yields 400,000,000 data points). After the full shell script (Fig. 4.5) has executed, the frequency response of the 80 MHz channel will be saved in eight .txt files as step_1.txt, step_2.txt, ... , step_8.txt.

Using the chirp signal transmission, handshaking synchronization protocol, and

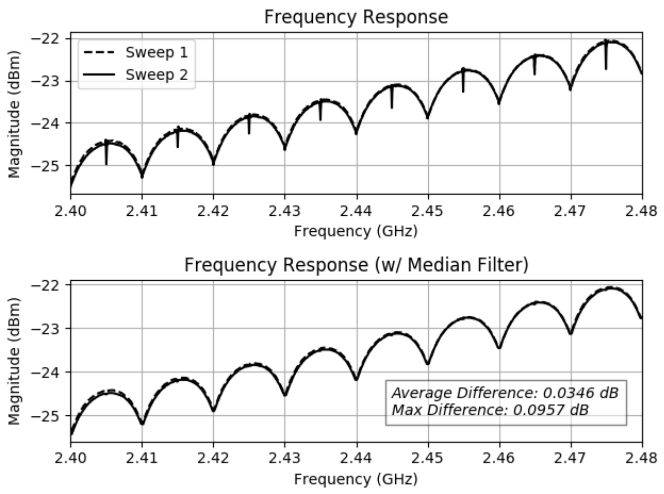


Figure 4.7: "Through" measurement taken over 2.4 GHz ISM band using chirp signal and handshaking protocol.

FFT frequency response calculation, the frequency response of a single wireless link can be measured in a distributed manner. The shell scripts presented in Fig. 4.4 and Fig. 4.5 are run on the same hardwired channel as the 10 MHz measurements used in Fig. 4.1. This 80 MHz wide measurement is made twice back-to-back to test repeatability, and displayed in Fig. 4.7 (raw data displayed in top plot). From the top plot it can be seen that there is an impulse at the center frequency of each 10 MHz measurement increment. This impulse is present but not nearly as severe in the 10 MHz measurements in Fig. 4.1, and is believed to be a product of switching the center frequency during the 80 MHz measurements. A common method in image processing used to filter impulsive noise is the median filter, which is popular due to its ability to filter impulsive noise while preserving edges [38]. The bottom plot in Fig. 4.7 shows the data filtered using a 5-point median filter, and it can be seen that the impulses at the center frequencies have been removed, while the over-all shape of the response is preserved. It will be shown later (Section 4.2) that even deep fades are accurately preserved by the median filter.

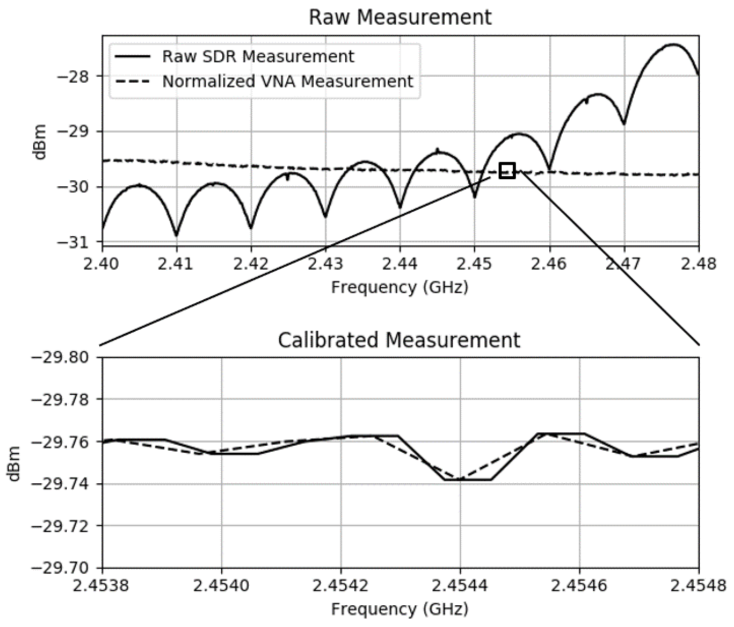


Figure 4.8: Comparing a measurement taken using a VNA in the hardwired environment to a raw SDR measurement (top) and a calibrated SDR measurement (bottom). The difference in scale between the top and bottom plots (4 dB vs 0.1 dB and 80 MHz vs 10 MHz) is important to note.

4.1.4 System Calibration

Fig. 4.7 shows that the system is repeatable to within ≤ 0.09 dB over 80 MHz of bandwidth. While it is unlikely that the true frequency response of the hardwired channel resembles the plot in Fig. 4.7, since the system is repeatable it should be possible to calibrate out the frequency response of the measurement system itself. The frequency response of the measurement system is found by measuring the frequency response of the hardwired "through" cable set-up using a VNA and comparing the VNA measurement to the measurement taken using the SDR system. This process is displayed in Fig. 4.8. The top plot displays a raw SDR measurement, and it can be seen that there seem to be peaks around the center frequencies and troughs around the edges of the measurement increments. Also displayed in the top plot is a VNA measurement taken over this band.

The frequency response of the measurement system is the difference between the SDR measurement and this VNA measurement, and once the frequency response of the system is subtracted from the SDR measurement, it matches the VNA measurement closely, evident in the bottom plot of Fig. 4.8. One thing to note is that the SDR measurement has 1025 measurement points and the VNA measurement has 551. So to find the response of the measurement system, each point of the SDR measurement is incremented over, and the closest point of the VNA measurement is used to compute the difference. This is evident in the bottom plot of Fig. 4.8, where it can be seen there are more points of measurement in the SDR measurement (solid line) than the VNA measurement (dotted line). Even so, the variations between the VNA measurement and calibrated SDR measurement are $\leq .005$ dB.

4.1.5 N-link Channel Measurements

A notable feature of the proposed measurement system is its ability to scale to measure multiple links simultaneously. This could be useful in wireless measurement campaigns, as the same system could be used to measure a single link, or N links depending on how many radios are being used. This feature could also be useful in wireless networks where this algorithm could be run on the devices in the network as a subroutine to characterize multiple links in the network simultaneously. In the proposed system, a single transmitter is used with N receiving radios, allowing for N links to be characterized. In a mesh network of nodes if the “roles” of each node alternated (i.e., which nodes were transmitting and receiving in the measurement system) this would allow for all links in the mesh network to be characterized.

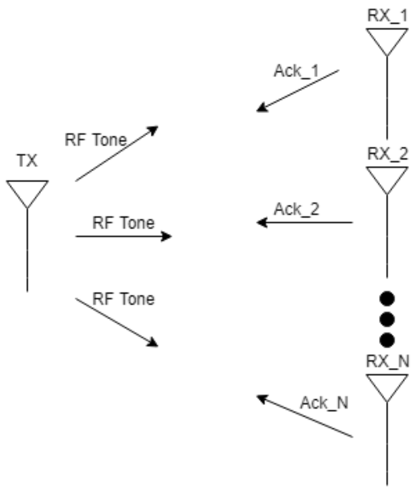


Figure 4.9: A schematic of the N-link channel measurement system being proposed.

With multiple distributed measurement links, synchronization of the system is vitally important. It is imperative that each of the radios switch their center frequencies simultaneously so that the entire transmitted chirp is captured at each receiving end of the system. In order to achieve this, the transmitter waits to receive an acknowledgement message from every receiver before incrementing its center frequency. To prevent collisions between packets being transmitted from the various receiving ends of the system, each measurement link is assigned a time slot over which to transmit the acknowledgement packets. This is a multiple access scheme called Time Division Multiple Access (TDMA) which allows for multiple communication links to be established on a single frequency channel. By utilizing TDMA communications, the transmitter can successfully wait to receive an acknowledgement packet from all of the receivers before moving to the next frequency.

4.2 System Validation

In Section 4.1 the measurement methodology of the system presented in this work was detailed, including the handshaking synchronization methods used in order to measure over distributed links. In this section, hardwired and wireless validating measurements will be presented. To validate measurements, both a VNA and the proposed SDR system will be used to measure the same links and their results will be compared.

When comparing the measurements of the SDR system to measurements from a VNA, it is important to remember what exactly each system is measuring. The S_{21} measurement of a VNA measures path-loss from one port of the system to the next. In other words, the frequency response it measures is in dB, and represents the ratio of received power to transmitted power. Since the SDR system being developed here is a distributed system, the exact transmit power is not assumed to be known by the receiver, and therefore it would not be possible to calculate this ratio. Instead the frequency response from the SDR measures the magnitude of the received power in dBm. In order to accurately compare these frequency responses, the VNA measurements will be normalized by the median value of the SDR measurement.

4.2.1 Hardwired

The simplest case of a hardwired link was already presented in Fig. 4.7 and Fig. 4.8 and is a direct link between transmitting end and receiving end. This hardwired case is used for system calibration in all measurements, but of more interest are links with multipath components, such as the Rayleigh channel displayed in Fig. 2.2 and the hyper-Rayleigh channel displayed in Fig. 2.3. In both of these scenarios, the transmitted signal reaches the receiver from multiple paths, each of which is delayed by a different amount. This delay is related to the length of the path the signal component traveled to reach the receiver. A hardwired simulation of a multipath environment can be accomplished by splitting the transmitted signal and having the signal components travel through different length cables before recombining at the receiver.

Fig. 4.10 displays a hardwired simulation of a Ricean communication channel, with a dominant LOS signal and a attenuated multipath signal (representing the lumped diffuse component). The hardwired two-signal path simulation displayed in Fig. 4.10 was measured using both the proposed SDR measurement system, and a Vector Network Analyzer. Both measurements are displayed in Fig. 4.11. It can be seen in this figure that this multipath creates some frequency selective fading, with a signal strength range of 14.6 dB over the 80 MHz band. Though fading is present in this channel, it is relatively benign compared to some of the more severe multipath environments that will be presented later in this chapter. Over this 80 MHz of measurement, and 14.6 dB range of signal strength, the mean-squared error (MSE) of the SDR measurements compared to VNA measurements is only 0.0042% σ

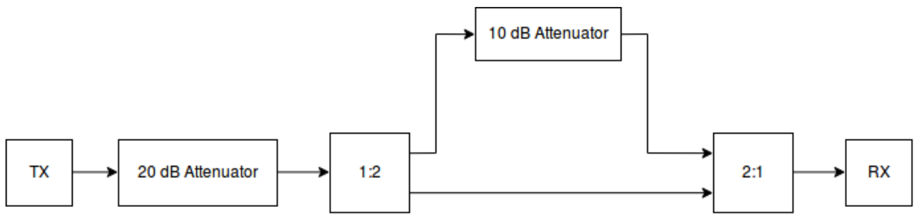


Figure 4.10: A hardwired simulation of a wireless channel with one LOS signal component, and one multipath component. The multipath component is delayed and attenuated compared to the LOS component.

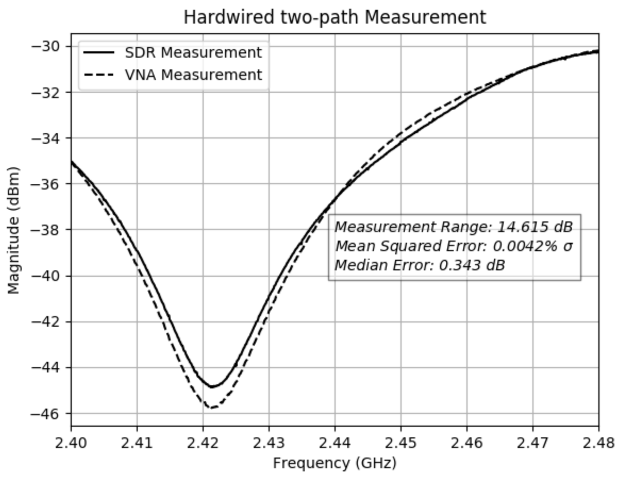


Figure 4.11: Frequency response measurements of the channel displayed in Fig 4.10 along with relevant error statistics. Note: the VNA measurement is normalized to the median value of thee SDR measurement for comparison between the curves.

(σ is the standard deviation of the VNA measurement) and the median error is only 0.342 dB.

To compute the median error, the difference between the SDR measurement and the nearest VNA measurement is computed at each SDR measurement point in dB, and the median of these errors is computed. The equation for computing MSE is displayed in equation 4.2 where n is the number of data points, Y_i is the observed data point (VNA measurement point), and $\hat{Y}_i$ is estimated data point (SDR measurement point).

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2 \tag{4.2}$$

In this case, the MSE is slightly more complicated to compute since the SDR system measurement has 1025 measurement points, while the VNA measurement has 551. To compute MSE every SDR measurement point is compared to the *nearest* VNA measurement point available. The Python function **calc_error** (displayed below) successfully calculates the raw MSE, the MSE in terms of the standard deviation, and the median error in dB. This function is used in all of the verification measurements to compare the frequency response measurements calculated using the SDR measurement system and using a VNA.

```python
def calc_error(sdr_dB, vna_dB):
    sdr_raw = 10**(sdr_dB/20.0)
    vna_raw = 10**(vna_dB/20.0)
    sdr_reso = len(sdr_raw)
    vna_reso = len(vna_raw)
    mse = 0
    db_err = np.zeros((sdr_reso))
    for i in range(0, sdr_reso):
        mse += abs(sdr_raw[i] - \
            vna_raw[int((i/float(sdr_reso))*float*vna_reso))])**2
        db_err[i] = abs(sdr_dB[i] - \
            vna_dB[int((i/float(sdr_reso))*float*vna_reso))])**2
    med_dB = np.median(db_err)
    mse = (mse / sdr_reso)
    mse_std = mse / np.std(err_raw)
    return mse, mse_std, med_dB
```

4.2.2 Wireless

After these hardwired measurements were taken, wireless measurements were taken with the SDR system and compared to VNA measurements. The reverberation chamber which was used to create the wireless channel in Fig. 2.5 is again utilized for these measurements to generate a highly multipath environment with severe frequency selective fading. The measurement set-up is displayed in Fig. 4.12 and the corresponding measurements in Fig. 4.13. It can be seen from these figures that this metallic chamber creates severe frequency selective fading over the channel with a signal strength range of ∼40 dB. It can also be seen that the SDR system performs remarkably well over this 80 MHz band with a MSE of only 0.0086% σ and a median

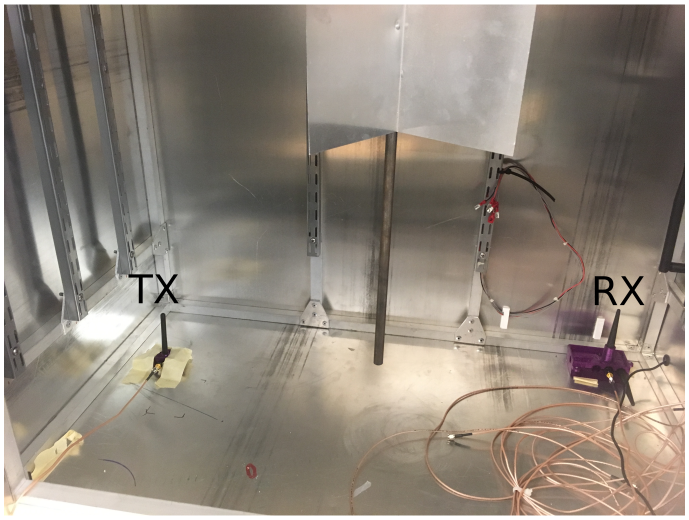


Figure 4.12: The single-link measurement taken within the reverberation chamber.

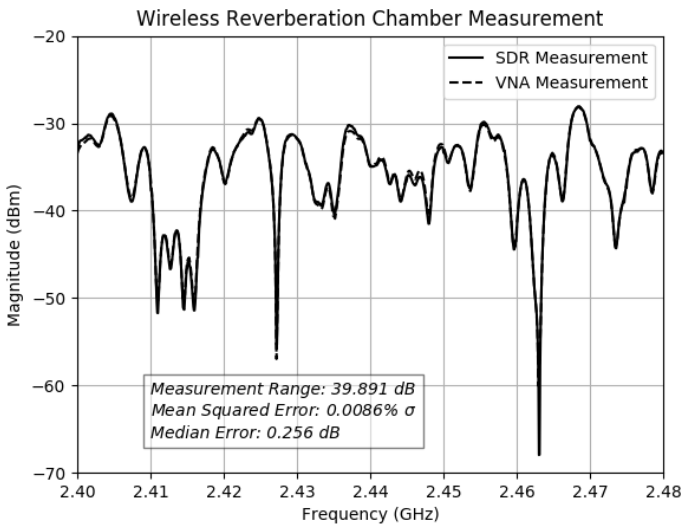


Figure 4.13: Frequency response measurements of the channel displayed in Fig 4.12.

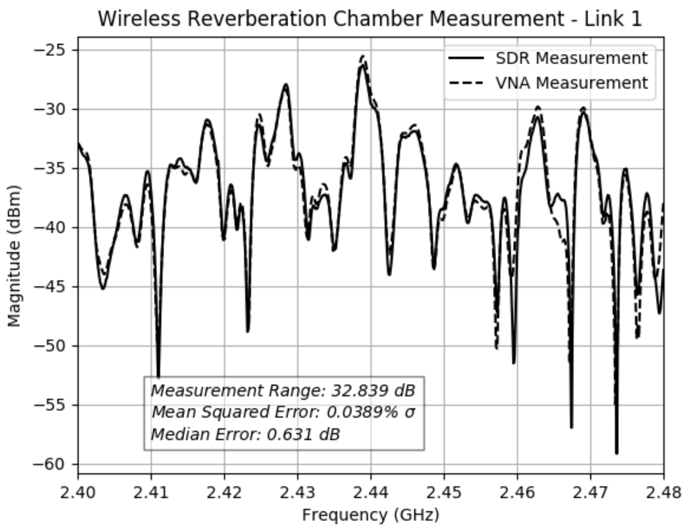


Figure 4.14: Link 1 of the two links measured simultaneously.

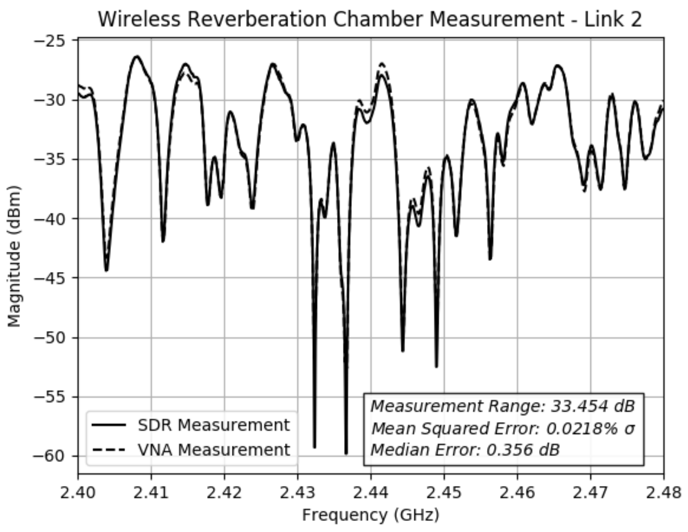


Figure 4.15: Link 2 of the two links measured simultaneously.

error of .256 dB.

The measurement presented in Fig. 4.13 was a single-link measurement, but as mentioned earlier in this chapter a novel aspect of this measurement system is its ability to scale to multiple links and measure them simultaneously. To test this, a simultaenous two-link measurement was set-up in the reverberation chamber and SDR measurements were again compared to VNA measurements. The results of the first link are displayed in Fig. 4.14 and the second link in Fig. 4.15 where it can be seen that the MSE with respect to σ are 0.0389% σ and 0.0218% σ respectively and the median errors are .631 dB and .356 dB respectively.

Often times, it is useful to view frequency response measurements statistically in order to get an idea of the *severity* of the fading present. In Fig. 4.16 the cumulative distribution functions (CDF) of each of the verification measurements are shown (the CDF was calculated using the measurements taken using the SDR measurement system). The CDF of the Rayleigh distribution (discussed in Section 2.2.2) is also plotted for reference. The hyper-Rayleigh region (from Section 2.2.3) is the region to the left of the Rayleigh curve and the Ricean region (from section 2.2.1) is the region to the right. It can be seen from Fig. 4.16 that the single-link measurement (orange line), and Link 2 of the two-link measurement (red) both exhibit approximately Rayleigh fading. Link 1 of the two-link measurement (green line) seems to exhibit less severe (i.e. Ricean) fading likely due to a stronger LOS signal component. The hardwired measurement (blue line) exhibits the most benign fading, evident both from Fig. 4.16 and its lower measurement range (14.6 dB). The hardwired measurement statistically exhibits traditional Ricean fading behavior, as expected from the hardwired set-up.

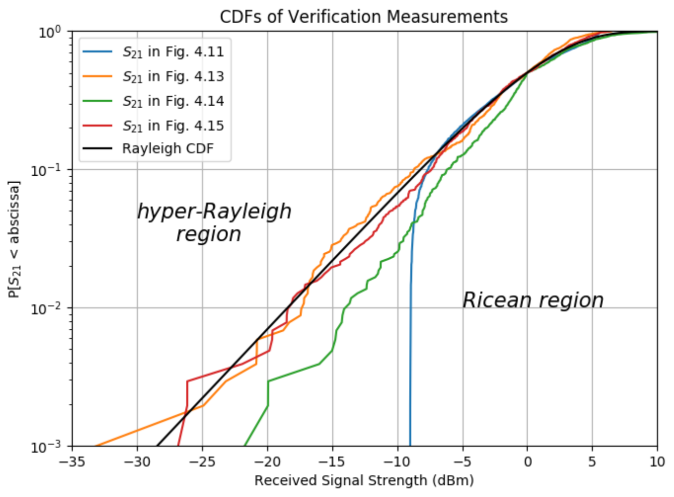


Figure 4.16: The Cumulative Distribution Functions (CDF) of each of the verification measurements presented in this section along with the CDF of the Rayleigh distribution plotted for reference.

Experiment	MSE (w/ respect to σ)	Median Error (dB)	Measurement Range
Hardwired	.0042 % σ	.343 dB	14.62 dB
Wireless - Single Link	.0086 % σ	.256 dB	39.89 dB
Wireless - Dual Link (Link 1)	.0389 % σ	.631 dB	32.84 dB
Wireless - Dual Link (Link 2)	.0218 % σ	.356 dB	33.45 dB

Table 4.1: A summary of the error statistics from the verification measurements presented in this chapter.

4.3 Conclusion

In this chapter, the operation of the proposed measurement system was detailed along with some verifying measurements to prove the systems accuracy. A summary of these results are displayed in Table 4.1. From Table 4.1 it can be seen that the proposed SDR measurement system is accurate to within a fraction of a percent of the standard deviation of the measurements, over *four* orders of magnitude of measurement range.

More specifically the average MSE of the proposed measurement system is 0.018 % σ operating on channels with measurement ranges reaching $\sim$ 40 dB, and the median error never exceeds 0.631 dB over this measurement range. Now that the accuracy of this measurement system has been demonstrated, the next chapter will focus on a possible application.

Chapter 5

Applications

In the prior section the operation of the proposed measurement system was introduced as well as validating measurements. It was shown that the system is able to accurately measure the scalar frequency response of a 80 MHz wireless channel to within an average MSE of 0.018 % σ and a median error within 0.631 dB with measurement ranges extending to $\sim$ 40 dB of signal strength variation. The system was also demonstrated to be able to measure multiple links simultaneously in a fully distributed manner. In this section a possible application of this measurement system is demonstrated.

5.1 Wireless Network in Cluttered Environment

As discussed in Chapter 2 the proposed measurement system is designed with small-scale signal propagation effects in mind. Such propagation effects are common in

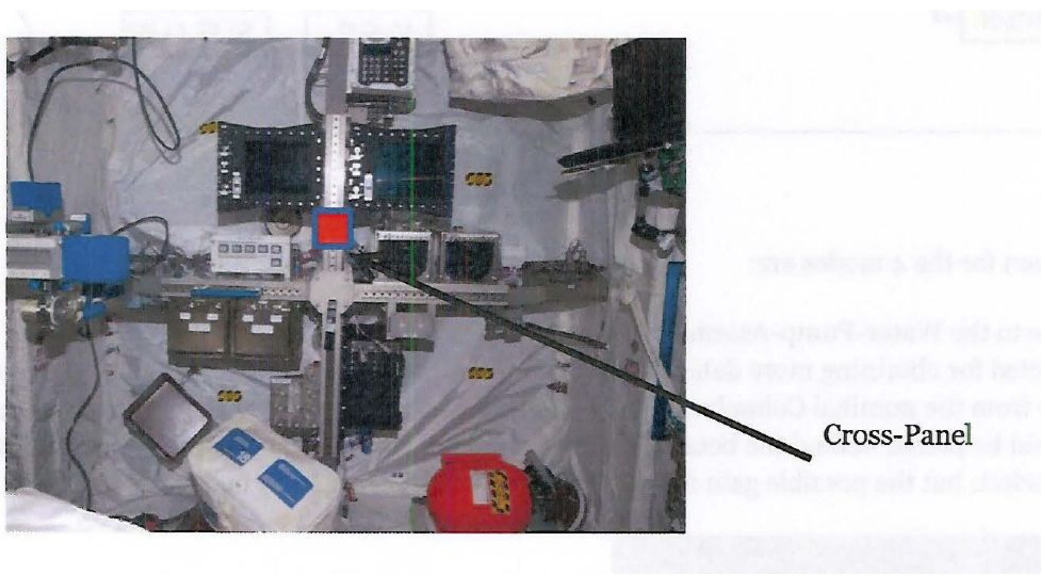


Figure 5.1: Placement of wireless sensor on metallic cross-panel aboard the ISS for the WISENET experiment. Retreived from [39].

reflective and cluttered environments which cause significant multipath, leading to severe frequency selective fading (as evident in Fig. 4.13, Fig. 4.14, and Fig. 4.15). This work aims to understand the channel conditions for low-power, low-bandwidth systems operating in such an environment. For example, a wireless sensor network (WSN) deployed for health monitoring purposes on a factory floor could expect significant frequency selective fading as a result of the metallic, reflective surfaces in the environment.

Highly reflective environments are also expected for emerging industrial machine-to-machine (M2M) applications. These effects have been shown both within a transport helicopter [3] and a Boeing 737-200 [4]. Such environments would also be expected for operation within spacecraft. Wireless sensor networks have been proposed for the International Space Station (ISS) [40] and extensive measurement campaigns have been undertaken to assess the feasibility of such systems [41]. NASA is in the process of undergoing more research on the subject of WSN on the ISS, for a project

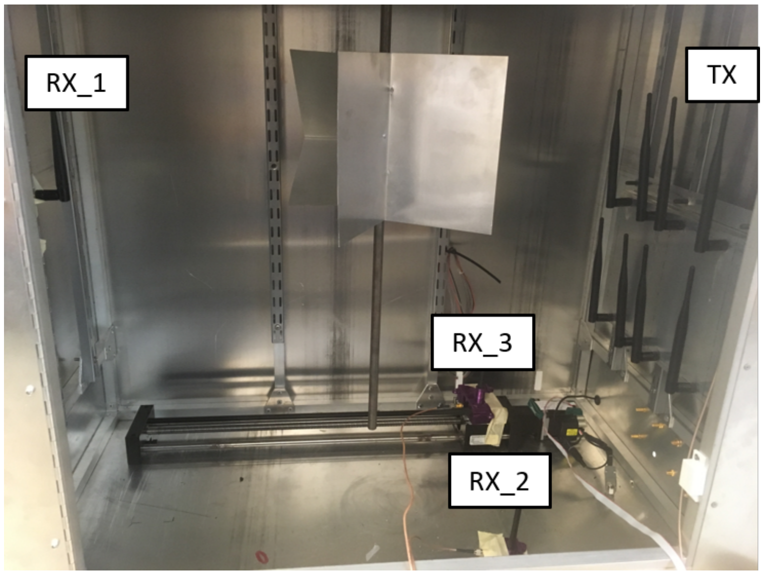


Figure 5.2: Three link propagation study within reverberation chamber.

called WISENET. Specifically in [42] it is said, "When operating, e.g., RF equipment within such a cage, reflections and interferences have to be taken into account." In this statement NASA is referring specifically to the multipath effects of the environment. Fig. 5.1 displays the placement of one of the wireless sensors aboard the ISS in the NASA study, it can be seen this this environment is indeed very cluttered and has numerous metallic components which could cause reflections.

5.1.1 Experiment Set-Up

This experiment is motivated by the scenario in which a WSN is operating in one of the aforementioned multipath environments and as one node in the network moves it alters the links between all elements of the network. The reverberation chamber used in Chapter 4 is again used here, a photo of the test set-up is displayed in Fig. 5.2. The transmitting antenna (TX) can be seen on the right side of the chamber, receiver one (RX_1) on the left side of the chamber, receiver two (RX_2) on the floor of the chamber, and receiver three (RX_3) on a linear track. The linear track is LabVIEW

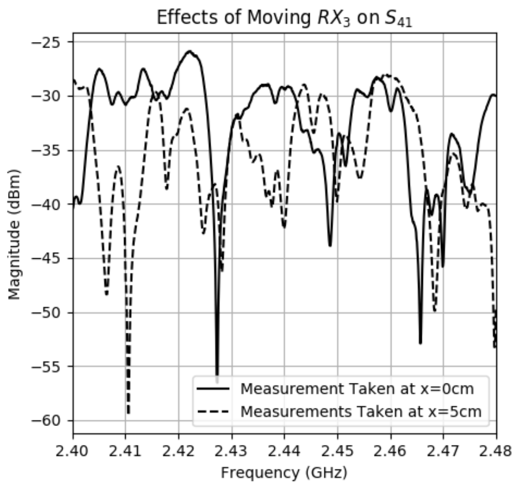


Figure 5.3: Two S_{41} ($TX->RX_3$) measurements taken before and after moving RX_3 5cm.

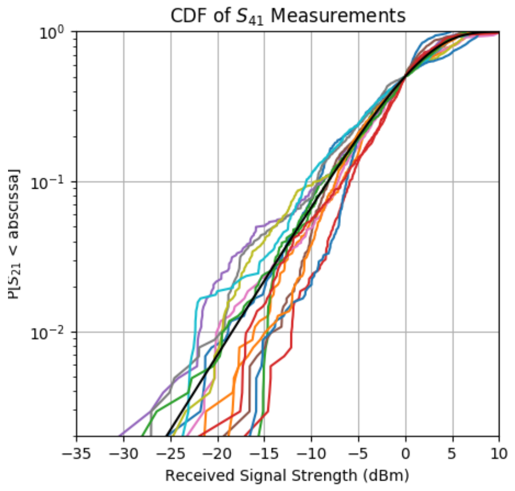


Figure 5.4: CDF of all Link 3 (S_{41}) measurements taken at 14 locations along the linear track. Black line is Rayleigh CDF plotted for reference.

controlled, and for this test moves to 14 different positions, in 5 cm increments ($\sim \frac{\lambda}{2}$). The antennas are connected to SDRs located outside of the reverberation chamber, and the system is run in a distributed manner from multiple computers. At each point on the track, the channel characterization algorithm is run on the three links simultaneously. The path loss measurements of the three links will be referred to as S_{21} (transmitter - link 1), S_{31} (transmitter - link 2), and S_{41} (transmitter - link 3).

5.1.2 Test Results

Small-scale fading is by definition fading that results from small changes in frequency or position. Changes due to small changes in frequency have already been examined in this thesis, but now small changes in position will be looked at as well. Fig. 5.3 shows how much the frequency response can change after a shift in position of approximately a half wavelength. These significant changes could cause issues for a wireless

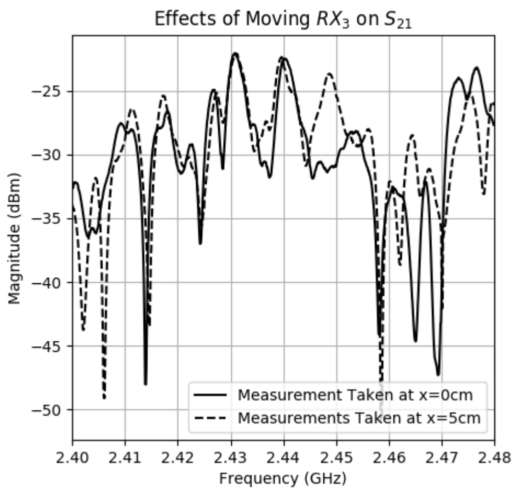


Figure 5.5: Two S_{21} ($TX- > RX_1$) measurements taken before and after moving RX_3 5cm.

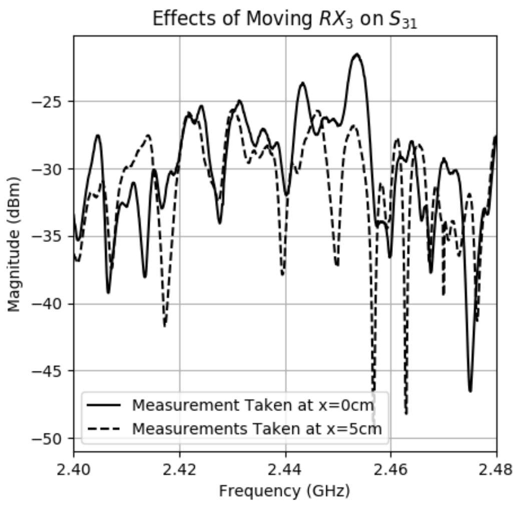


Figure 5.6: Two S_{31} ($TX- > RX_2$) measurements taken before and after moving RX_3 5cm.

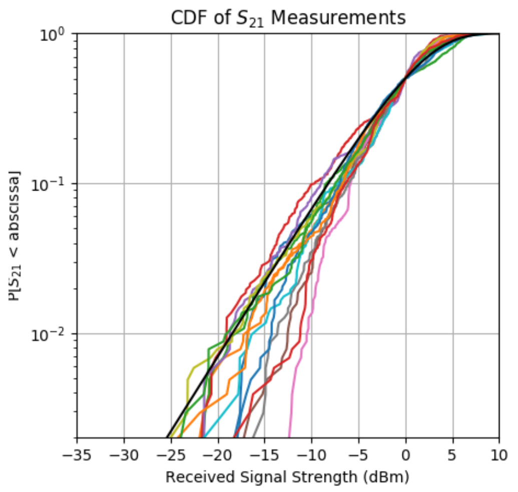


Figure 5.7: CDF of all Link 1 (S_{21}) measurements taken at 14 locations along the linear track. Black line is Rayleigh CDF plotted for reference.

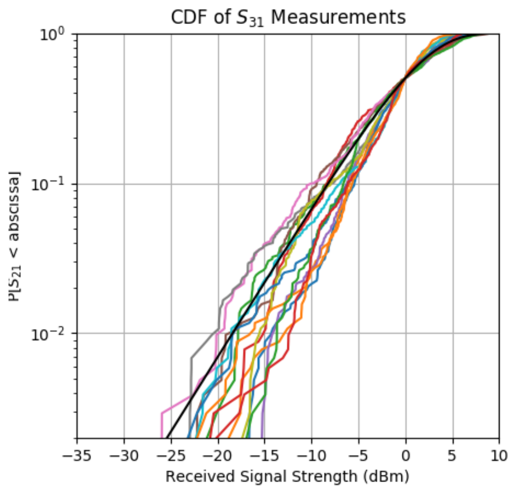


Figure 5.8: CDF of all Link 2 (S_{31}) measurements taken at 14 locations along the linear track. Black line is Rayleigh CDF plotted for reference.

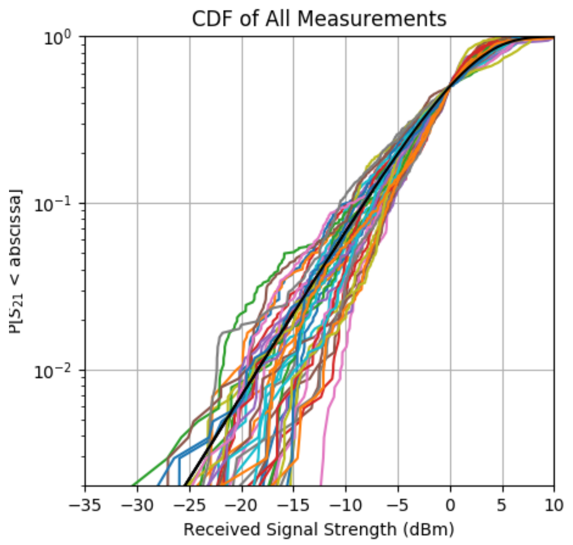


Figure 5.9: Cumulative distribution functions of all 42 (14 measurements over three links) measurements taken in this experiment, with the CDF of the Rayleigh distribution plotted for reference.

device operating over this bandwidth as frequencies which were usable at the original position are now not usable, and vice-versa. Looking at all 14 Link 3 measurements statistically in Fig. 5.4, it can be seen that while most measurements seem to resemble the Rayleigh distribution, there is a wide range of statistical severities of fade. In other words, some locations on the track may be “better” to communicate at than others. If sensors in a WSN ran this algorithm, or a similar one, to characterize the environment around them then real-time decisions could be made about the optimal frequency and/or position to operate at.

One could likely infer that moving a node in a wireless network would alter the characteristics of the link between that node and another. But what is more interesting is the fact that in this highly multipath environment, changing the position of RX_3 will not only affect the link between TX and RX_3, but will also significantly alter the link between TX and RX_1 or RX_2. This is due to severity of the small-scale effects in this environment. Fig. 5.5 and Fig. 5.7 show significant changes in

	Ricean	Approximately Rayleigh	hyper-Rayleigh
Link 3	6 (42.9 %)	2 (14.3 %)	6 (42.9 %)
Link 1	11 (78.5 %)	1 (7.1 %)	2 (14.3 %)
Link 2	8 (57.2 %)	4 (28.6 %)	2 (14.3 %)
All Links	25 (59.5 %)	7 (16.7 %)	10 (23.8 %)

Table 5.1: A summary of the fade-types of all measurements taken in this experiment.

link 1 due to 5cm changes in the position of RX_3, and Fig. 5.6 and Fig. 5.8 show the same for link 2. From this data it is clear that as one single node in a network moves throughout a cluttered environment, it becomes important to simultaneously be characterizing all possible communication links as all of the links will be effects from these movements.

Fig. 5.9 shows the wide range of channel scenarios that were measured in this experiment and Table 5.1 displays the percentage of various multipath scenarios present throughout this experiment. It was determined whether a measurement was Ricean, approximately Rayleigh, or hyper-Rayleigh using the 10 % fade-depth (10%FD) parameter [43]. The 10%FD is the point at which 10% of the channel data is more severe, and 90% is less severe. This metric for Rayleigh fading is approximately -8.2 dB, and channels whose 10%FD fell within ±0.5 dB were considered approximately Rayleigh, while lower values were labeled hyper-Rayleigh and higher values were labeled Ricean. It can be seen from Table 5.1 and Fig. 5.9 that there is a wide range of possible channel scenarios present in this four node, three link, network that results from moving one node 70 cm. An algorithm which could be run alongside existing functionality on a wireless node which employs SDR would be extremely useful. It could potentially allow a network to identify optimal operating points and adjust accordingly.

5.2 Conclusion

This chapter presented a case for a possible application of the proposed SDR based channel measurement system. These results are reflective of the channels a network would be expected to operate over within the environments discussed at the beginning of this chapter. Through a wireless measurement campaign using the proposed SDR system it was shown that all links in a three link network were significantly altered by moving a single receiver in 5 cm increments. From these measurements, decisions can be made about optimal node placements and frequencies of operation. An argument is made for the usefulness of such a channel measurement algorithm as a means of characterizing the links in a network of wireless devices alongside other functionality as a means of making decisions about network operation.

Chapter 6

Conclusions and Future Work

6.1 Contributions

This work aimed to develop a novel channel measurement system using a software-defined radio (SDR) platform. The system's design and performance has been presented throughout this thesis. As discussed in Section 3.4 on prior work, SDR-based channel measurement systems are an emerging area of research and there are a few examples in the literature of such systems. Most of these systems focus on time-domain measurements and use SDRs whose operational bandwidth exceed the bandwidth of the channel they are measuring. In addition to this these systems are not always distributed. The proposed system measures scalar frequency response over a 80 MHz bandwidth (utilizing 10 MHz of repeatable operational bandwidth) in a completely distributed manner. Specifically, the major contributions of the system are as follows:

1. *Development of a technique for measuring the scalar frequency response of a wireless channel using SDR and a "chirped" tone.*

In Section 4.1.1 the method of measuring scalar frequency response in the proposed system was discussed. The system transmits a tone at incremental frequencies measuring the magnitude of the tone at each point, effectively sampling the frequency response in the frequency domain.

2. *Development of an algorithm for the extension of synchronized measurement beyond the SDR operational bandwidth for multiple distributed measurement links.*
   As mentioned above, the repeatable operational bandwidth of the SDR system is found to be 10 MHz, so the center frequencies of the SDRs must be changed eight times throughout a 80 MHz measurement. A frequency stepping and handshake protocol presented in Section 4.1.2 was developed to synchronize the changing of center frequencies between measurement ends. This algorithm was extended in Section 4.1.5 to allow synchronization across multiple simultaneous measurement links utilizing Time Division Multiple Access (TDMA) techniques to transmit handshake messages at different time-slots for various receiving ends of the system. To the author's knowledge, these algorithms are significantly different than anything in the literature being used in SDR based channel measurement systems.

3. *Demonstration of the proposed SDR based system when compared to measurements taken using a Vector Network Analyzer (VNA) to demonstrate the systems accuracy.*
   Measurements presented in Section 4.2 demonstrate the system's ability to measure channels with a mean-squared error of 0.018% σ and a median error not exceeding 0.631 dB when compared to VNA measurements. These measurements were taken over a 80 MHz bandwidth with a signal-strength range of

$\sim$40 dB.

6.2 Future Work

There are several possible directions this work could be taken in the future. In this section, possible future work is discussed. Specifically, future work involving extending the measurement system to a vector measurement system, and a case specific frequency-doubling measurement system are discussed.

6.2.1 Phase Measurements

As mentioned in Section 2.3.2, in this particular application the scalar frequency response is arguably the most valuable information about the wireless channel. This is because signal reliability is heavily dependent on the signal-to-noise ratio (SNR) and a low magnitude signal is problematic. In some applications phase information can be important as well. For example, when both the time and frequency domain information on the channel are desired the phase information is needed to calculate impulse response from frequency response.

Due to the system being completely distributed, the local oscillators (LOs) used at the receiving and transmitting ends of the system are inherently asynchronous. Without these LOs being synchronized, it will not be possible to get accurate phase measurements. There exist methods of synchronizing the clocks of SDRs wirelessly (e.g., available as extension to USRP SDRs [44], claims 25 ppb accuracy by "locking too" GPS antenna) and as such we had hoped to add optional phase measurements to our system. The clock of the HackRF radio are accurate to within $\pm$ 10 ppm and the

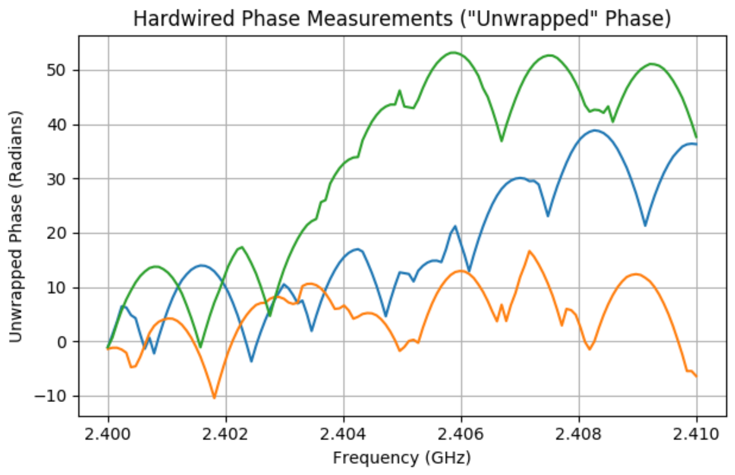


Figure 6.1: Three phase measurements taken in a hardwired environment over 80 MHz bandwidth.

BladeRF clock accurate to within ± 1 ppm, so for phase measurements two BladeRF would be used. The system in [36] presented vector measurements (i.e., magnitude and phase) over a 43.5 MHz bandwidth using two BladeRF, but their methods of calibration to achieve repeatable measurements are not clear.

The BladeRF have an option to be configured for MIMO (multiple-input multiple-output) communications by synchronizing the clocks of multiple radios. This is done by connecting the clock pins on the board, and configuring one radio to be "master" and the other to be "slave." In this way, the LO on the master device drives the LO on the slave device. Two BladeRF radios were configured in this manner and the frequency response was calculated using the methods discussed in Section 4.1, but this time instead of just calculating the FFT magnitude at each frequency bin, the FFT phase was calculated as well. This test was run three times on a hardwired set-up both with and without the clock configuration discussed above, and the results are displayed in Fig. 6.1.

Fig. 6.1 displays an attempt at extending the scalar measurements presented in this thesis to a vector measurement system which also measures phase. The phase data was “unwrapped” using the Python’s NUMPY library, but as seen are not repeatable. A possible cause of this is that even though the clocks are synchronized to within ± 1ppm, there is still a variable phase offset that changes randomly between measurements preventing them from being repeatable. Reconciling this phase offset may lead to repeatable phase measurements.

6.2.2 Frequency Doubling Measurements

A benefit of a channel measurement system based in software is that the system can be extended and customized to certain measurement scenarios relatively easily. One such example in a multidisciplinary UVM project is a system which could be developed for the study in [45]. In this paper, wireless sensor interrogation is studied using a passive frequency doubling reflectenna which transmits a signal at double the frequency of the received signal. Specifically, a 1.28 GHz tone is transmitted towards the reflectenna and a 2.56 GHz tone is received. In this particular study, the reflectenna is buried at varying depths in soil in order to investigate the effects of soil characteristics on signal propagation; a schematic of the test set-up is displayed in Fig. 6.2.

The tasks of the signal generator (SG in Fig. 6.2) and spectrum analyzer (SA in Fig. 6.2) could be replaced by this SDR system with minimal modifications. It would in theory be possible to customize this measurement so that it transmits a signal at frequency f and measures the signal at frequency $2f$. This would allow for a more reconfigurable testing system (i.e. measuring at different frequency pairs than 1.28 GHz / 2.56 GHz) and would also allow for frequency bands to be swept over. For this specific project it is desired to implement this wireless interrogator on a drone,

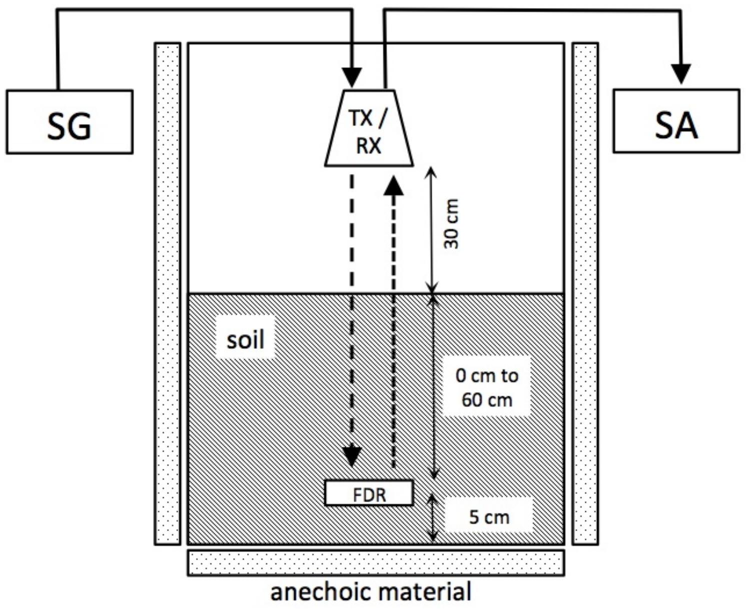


Figure 6.2: The test set-up used in [45].

and therefore there would be other wireless communication duties of the interrogator beyond just the frequency doubling measurements. An SDR based interrogator is a good candidate for this application as SDR is highly configurable, and the same hardware used for the measurements could be used for other communication needs with relative ease.

Preliminary testing of the system for this application has shown that the SDR system *can* detect 2.56 GHz signals as low as -100 dBm, which will be imperative to the system working for this application. Testing has also shown that when transmitting a 1.28 GHz signal at -10 dBm using the SDR system, a significant harmonic occurs at 2.56 GHz which interferes with the measurements. The next step in this project is to mitigate this harmonic either with modifications to BladeRF firmware or with an external filter.

6.3 Final Words

Measuring wireless propagation in an environment gives insights into *what* systems will be able to operate in a particular environment as well as *how* these systems will operate. This information is vitally important to wireless applications in highly reflective environments, such as those which are to be expected in emerging machine-to-machine applications. Many devices that will operate in these environments will be low-power, low-bandwidth devices (e.g. wireless sensor networks adhering to IEEE 802.15.4 specifications). Therefore it is especially useful to take propagation measurements in these environments as to determine optimal device locations and optimal frequency channel selection. An example of such a scenarios is the scenario presented in Chapter 5 where the wireless links between a network of four nodes (one TX, three RX) was continuously characterized as one of the nodes in the network moved. In this chapter it was seen that the information gained from these measurements can be very useful when operating a network in a highly reflective environment.

This work presents a novel, low-cost, distributed method of taking these wireless channel measurements. The system has the ability to make channel measurement campaigns more accessible as it is more portable than typical channel measurement devices (e.g. vector network analyzers), and also has the ability to be run alongside existing functionality on wireless devices which already employ SDRs. Using this algorithm, a node in a wireless network such as the one presented in Chapter 5 could be periodically characterizing the environment around it and making real-time decisions on node placement, antenna configurations, etc. This work provides the proof-of-concept of a low-cost wireless channel measurement system based in software-defined radio.

Bibliography

[1] A. Meola, "What is the Internet of Things (IoT)? Meaning & Definition," *Business Insider* [Online] Available: https://www.businessinsider.com/internet-of-things-definition. [Accessed: July 20, 2018]

[2] K. Gordon, "M2M (machine-to-machine) - Statistics & Facts," *Statista: The Statistics Portal* [Online] Available: https://www.statista.com/topics/1843/m2m-machine-to-machine. [Accessed: July 20, 2018]

[3] R. Ketcham, J. Frolik, J. Covell, "Propagation Measurements and Statistical Modeling for Wireless Sensor Systems Aboard Aircraft," *IEEE Trans. on Aerospace and Electronic Systems*, vol. 44, pp. 1609-1615, October 2008

[4] J. Chuang, N. Xin, H. Huang, S. Chiu, and D. Michelson, "UWB Radiowave Propagation within the Passenger Cabin of a Boeing 737-200 Aircraft," In Proc. IEEE 65th Vehicular Technology Conference, Spring 2007

[5] J. Frolik, "Deciwavelength-Scale Fade Mitigation," In Proc. IEEE European Conference on Antennas and Propagation, 2014

[6] J. Frolik, "A Case for Considering Hyper-Rayleigh Fading Channels", *IEEE Trans. on Wireless Communications*, vol. 6, pp. 1235-1239, April, 2007

[7] J. Frolik, T. Weller, S. DiStasi, J. Cooper, "A Compact Reverberation Chamber for Hyper-Rayleigh Channel Emulation," *IEEE Trans. Antennas Propagation*, vol. 57, pp. 3962-3968, December, 2009.

[8] J. Jamison, B. Hewgill, J. Frolik, "A Configurable and Repeatable Multipath Channel Emulator," In Proc. IEEE Antennas and Propagation Society Meeting, July 2017

[9] T. Rappaport, *Wireless Communications: Principle and Practice 2nd Ed* Upper Saddle River NJ, Prentice Hall PTR 2001

[10] Bernard Skyler, “Rayleigh Fading Channels in Mobile Digital Communication Systems Part I: Characterization,” *IEEE Communications Magazine* vol. 35, Issue 9, September 1997

[11] S. Kay, *Intuitive Probability and Random Processes Using MATLAB* 4 ed., Springer Science+Business Media LLC 2006

[12] G. Durgin, T. Rappaport, D. de Wolf, “New Analytical Models and Probability Density Functions for Fading in Wireless Communications,” *IEEE Trans. on Communications*, vol. 50, pp. 1005-1015, Aug. 2002

[13] T. Rappaport, “Characterization of UHF Multipath Radio Channels in Factor Buildings”, *IEEE Trans. on Antennas and Propagation*, vol. 37, pp 1058-1069, August 1989

[14] T. Rappaport, S. Seidel, R. Singh, “900-MHz Multipath Propagation Measurements for U.S. Digital Cellular Radiotelephone,” *IEEE Trans. on Vehicular Technologies*, vol. 39, pp 132-139, May 1990

[15] Cox D. C., “Delay Doppler Characteristics of Multipath Delay Spread and Average Excess Delay for 910 MHz Urban Mobile Radio Paths,” *IEEE Trans. on Antennas and Propagation*, vol. 20, pp 625-635, September 1972

[16] J.-J. van de Beek, O Edfors, M. Sandell, S.K. Wilson, P.O. Borjesson, “On channel estimation in OFDM systems,” In Proc. IEEE Vehicular Technology Conference, July 1995

[17] B.P. Lathi, Z. Ding, *Modern Digital and Analog Communication Systems*, 4 ed., Oxford University Press 2010

[18] 802.15.4-2015 - IEEE Standard for Low-Rate Wireless Networks, ICS: 35.110 - Networking

[19] “ZigBee Specification FAQ,” *ZigBee Alliance* [Online], Available: https://web.archive.org/web/20130627172453/http://www.zigbee.org/Specifications /ZigBee/FAQ.aspx

[20] “A short history of software-defined radio (SDR) technology”, *nutaq.com* [Online], Available: https://www.nutaq.com/blog/short-history-software-defined-radio-sdr-technology

[21] G.D. Space Systems Technology Group, “New Research Lab Leads to Unique Radio Receiver,” *E-Systems Team Magazine*, vol. 5, pp. 6-7, 1985

[22] R.J. Lackey and D.W. Upmal, “Speakeasy: The Military Software Radio,” *IEEE Communications Magazine*, vol. 33, pp. 56-61, May 1995

[23] J. Mitola, “Software Radios: Survey, Critical Evaluation and Future Directions,” *IEEE Aerospace and Electronic Systems Magazine*, vol. 8, pp. 25-36, April 1993

[24] Oppenheim, Schafer, *Discrete-time Signal Processing 3rd Ed* Prentice Hall 2009

[25] A. Vaccari, EE 275. Class Lecture, Topic: “Digital Processing of Analog Signals,” Department of Electrical and Biomedical Engineering, The University of Vermont, Burlington Vermont, Spring 2018

[26] Machado-Fernández, José Raúl. (2015). “Software Defined Radio: Basic Principles and Applications”. Facultad de Ingeniería, 24(38), 79-96. [Online] Available: http://www.scielo.org.co/scielo.php?script=sci_arttext&pid=S0121-11292015000100007&lng=en&tlng=en. [Accessed: June 18, 2018]

[27] “Software Defined Radio Receiver USB Stick - RTL2832 w/R820T,” *adafruit.com*, [Online] Available: https://www.adafruit.com/product/1497

[28] “Great Scott Gadgets HackRF One - Software Defined Radio,” *adafruit.com*, [Online] Available: https://www.adafruit.com/product/3583

[29] “bladeRF x40,” *sparkfun.com* [Online] Available: https://www.sparkfun.com/products/14041

[30] “USRP X310,” *ettus.com* [Online] Available: https://www.ettus.com/product/details/X310-KIT

[31] “About GNU Radio,” *gnuradio.org* [Online] Available: https://www.gnuradio.org/about/

[32] N. Hosseini, D. Matolak, “Wide Band Channel Characterization for Low Altitude Unmanned Aerial System Communication Using Software Defined Radio,” In Proc. Integrated Communications, Navigation, Surveillance Conference (ICNS), 2018

[33] H. Boeglen, A. Traore, M. Peinado, R. Lefort, and R. Vauzelle, “An SDR Based Channel Sounding Technique for Embedded Systems,” In Proc. IEEE 11th European Conference on Antennas and Propagation (EUCAP) 2017

[34] H. Mukhtar, A. Al-Dweik, M. Masouridis, and T. Stouraitis, “Performance Evaluation of Time-Domain Interleaved OFDM Systems using Software Defined Radio Platforms,” In Proc. IEEE International Conference on Electrical and Computing Technologies and Applications (ICECTA) 2017

[35] J. Li, Y. Zhao, C. Tao, and B. Ai, "System Design and Calibration for Wideband Channel Sounding With Multiple Frequency Bands," *IEEE Access* (vol 5) 2017

[36] T. Srisooksai, J. Takada, and K. Saito, "Portable Wide-band Channel Sounder Based Software Defined Radio for Studying The Radio Propagation in An Outdoor Environment," In Proc. IEEE International Symposium on Antennas and Propagation (ISAP) 2017

[37] S. K. Rakesh, "Gaussian Frequency-shift Keying With GNU Radio," *scribd.com* [Online] Available: https://www.scribd.com/doc/254559988/Gaussian-Frequency-shift-Keying-With-GNU-Radio [Accessed: June 29, 2018]

[38] A. Vaccari, EE 275. Class Lecture, Topic: "Nonlinear Digital Filters," Department of Electrical and Biomedical Engineering, The University of Vermont, Burlington Vermont, Spring 2018

[39] "WiSe-Net - WIrelessSEnsor-NETwork (DLR)" *eea.spaceflight.esa.int/portal/* [Online] Available: http://eea.spaceflight.esa.int/portal/exp/?id=9448 [Accessed: July 5, 2018]

[40] H.J. Beestermöller, H.-J. Borchers, H. Luttmann, J. Sebald, M.-C. Sinnreich, M. Schneider and V. Schmid, "Wireless-Sensor Networks in Space Technology Demonstration on ISS", in Proc. Dresdner Sensor-Symposium 2015, Dresden, Germany, Dec 7-9 2015

[41] M. Drobczyk, C. Strowik. C. Philpot, "A Wireless Communication and Positioning Experiment for the ISS Based on IR-UWB", in Proc. IEEE 2017 Wireless Communications and Networking Conference (WCNC) 2017

[42] "WISENET (WISENET) - 03.21.18," *nasa.gov* [Online] Available: https://www.nasa.gov/mission_pages/station/research/experiments/1775.html [Accessed: July 5, 2018]

[43] J. Frolik, "A Practical Metric for Fading Environments", *IEEE Wireless Communications Letters*, Vol. 2, No. 2, April 2013.

[44] "Board Mounted GPSDO (OCXO) Recommended for USRP X300/X310" *ettus.com* [Online] Available: https://www.ettus.com/product/details/GPSDO-MINI [Accessed: August 6, 2018]

[45] J. Frolik, J. Lens, M. Dewoolkar, and T. Weller, "Effects of Soil Characteristics on Passive Wireless Sensor Interrogation," *IEEE Sensors Journal*, Vol. 18, No. 8, April 2018

Appendix A

Source Code

This appendix includes the source code for the operation of the measurement system discussed throughout this thesis.

A.1 Transmitter

This section includes the code written for the transmitting end of the measurement system. Code is included that is used for the transmitter, in a multi-link measurement system.

A.1.1 Shell Script

```bash
#!/bin/bash

echo "2.4 GHz ISM Frequency Sweep - Transmitter"

for ((i = 0; i < 8; i = i+1))
    do
        python TX_threelink.py -c 2.4${i}5e9 -a 10
        echo "wait..."
    done

echo "Done"
```

A.1.2 Python Code

```python
#!/usr/bin/env python2
# -*- coding: utf-8 -*-
```

```
##################################################
# GNU Radio Python Flow Graph
# Title: Chirp Tx
# Generated: Mon Apr 23 11:07:02 2018
##################################################

from gnuradio import analog
from gnuradio import blocks
from gnuradio import digital
from gnuradio import eng_notation
from gnuradio import filter
from gnuradio import gr
from gnuradio.eng_option import eng_option
from gnuradio.filter import firdes
from optparse import OptionParser
import osmosdr
import threading
import time
import numpy as np
import os


class chirp_tx(gr.top_block):

    def __init__(self):

    # Command line parsing
    parser = OptionParser()
    parser.add_option("-c", "--center", dest="center_freq",
        help="Desired Center Frequency", default=2.405e9)
    parser.add_option("-a", "--amplitude", dest="amplitude",
        help="Signal Amplitude (of Sine Source)", default=1)
    parser.add_option("-e", "--end", dest="end",
        help="End program when sweep finishes (instead of waiting for ack)",
        default=False, action="store_true")
    (options, args) = parser.parse_args()


        gr.top_block.__init__(self, "Chirp Tx")

        ##################################################
        # Variables
        ##################################################
        self.variable_function_probe_0 = variable_function_probe_0 = 0
        self.start_freq = start_freq = float(options.center_freq)
        self.samp_rate = samp_rate = 20e6
        self.freq = freq = 0
    self.ampl = ampl = float(options.amplitude)
    self.req_ampl = float(options.amplitude)
    self.end = options.end

        ##################################################
        # Blocks
        ##################################################
        self.square = blocks.probe_signal_f()
        def _variable_function_probe_0_probe():
            while True:
                val = self.square.level()
                try:
                    self.set_variable_function_probe_0(val)
                except AttributeError:
                    pass
                time.sleep(1.0 / (10))
        _variable_function_probe_0_thread = threading.Thread(target =
        _variable_function_probe_0_probe)
        _variable_function_probe_0_thread.daemon = True
        _variable_function_probe_0_thread.start()
        self.osmosdr_sink_0 = osmosdr.sink( args="numchan=" + str(1) + " "
        + "hackrf=391890cf" )
        self.osmosdr_sink_0.set_time_now(osmosdr.time_spec_t(time.time()),
         osmosdr.ALL_MBOARDS)
        self.osmosdr_sink_0.set_sample_rate(samp_rate)
        self.osmosdr_sink_0.set_center_freq(start_freq, 0)
        self.osmosdr_sink_0.set_freq_corr(0, 0)
        self.osmosdr_sink_0.set_gain(6, 0)
        self.osmosdr_sink_0.set_if_gain(0, 0)
        self.osmosdr_sink_0.set_bb_gain(0, 0)
        self.osmosdr_sink_0.set_antenna("", 0)
        self.osmosdr_sink_0.set_bandwidth(0, 0)

        self.blocks_throttle_0 = blocks.throttle(gr.sizeof_float*1,
        samp_rate,True)
        self.analog_sig_source_x_1 = analog.sig_source_f(samp_rate,
        analog.GR_SQR_WAVE, 5, 1, 0)
        self.analog_sig_source_x_0 = analog.sig_source_c(samp_rate,
        analog.GR_SIN_WAVE, freq, self.ampl, 0)

        ##################################################
        # Connections
        ##################################################
        self.connect((self.analog_sig_source_x_0, 0), (self.osmosdr_sink_0, 0))
        self.connect((self.analog_sig_source_x_1, 0),
        (self.blocks_throttle_0, 0))
        self.connect((self.blocks_throttle_0, 0), (self.square, 0))

    def get_variable_function_probe_0(self):
        return self.variable_function_probe_0

    def set_variable_function_probe_0(self, variable_function_probe_0):
        self.variable_function_probe_0 = variable_function_probe_0

    def get_start_freq(self):
        return self.start_freq

    def set_start_freq(self, start_freq):
        self.start_freq = start_freq
        self.osmosdr_sink_0.set_center_freq(self.start_freq, 0)

    def get_samp_rate(self):
        return self.samp_rate

    def set_samp_rate(self, samp_rate):
        self.samp_rate = samp_rate
        self.analog_sig_source_x_0.set_sampling_freq(self.samp_rate)
        self.analog_sig_source_x_1.set_sampling_freq(self.samp_rate)
        self.blocks_throttle_0.set_sample_rate(self.samp_rate)
        self.osmosdr_sink_0.set_sample_rate(self.samp_rate)

    def get_freq(self):
        return self.freq

    def set_freq(self, freq):
        self.freq = freq
        self.analog_sig_source_x_0.set_frequency(self.freq)

    def get_ampl(self):
    return self.ampl

    def set_ampl(self, ampl):
    #self.ampl = ampl
    self.analog_sig_source_x_0.set_amplitude(ampl)

class handshake_RX(gr.top_block):

    def __init__(self):
        gr.top_block.__init__(self, "Handshake Rx")

        ##################################################
        # Variables
        ##################################################
        self.variable_function_probe_1 = variable_function_probe_1 = 0
        self.samp_rate = samp_rate = 10e6

        ##################################################
        # Blocks
        ##################################################
        self.data_probe = blocks.probe_signal_vf(1024)
        def _variable_function_probe_1_probe():
            while True:
                val = self.data_probe.level()
                try:
                    self.set_variable_function_probe_1(val)
                except AttributeError:
                    pass
                time.sleep(1.0 / (1e6))
        _variable_function_probe_1_thread = threading.Thread(target =
        _variable_function_probe_1_probe)
        _variable_function_probe_1_thread.daemon = True
        _variable_function_probe_1_thread.start()
        self.osmosdr_source_0 = osmosdr.source( args="numchan=" + str(1) + " "
         + "hackrf=383f27cf" )
        self.osmosdr_source_0.set_time_now(osmosdr.time_spec_t(time.time()),
        osmosdr.ALL_MBOARDS)
        self.osmosdr_source_0.set_sample_rate(samp_rate)
        self.osmosdr_source_0.set_center_freq(302e6, 0)
        self.osmosdr_source_0.set_freq_corr(0, 0)
        self.osmosdr_source_0.set_dc_offset_mode(0, 0)
        self.osmosdr_source_0.set_iq_balance_mode(0, 0)
        self.osmosdr_source_0.set_gain_mode(True, 0)
        self.osmosdr_source_0.set_gain(14, 0)
        self.osmosdr_source_0.set_if_gain(25, 0)
        self.osmosdr_source_0.set_bb_gain(30, 0)
        self.osmosdr_source_0.set_antenna("", 0)
        self.osmosdr_source_0.set_bandwidth(0, 0)

        self.low_pass_filter_0 = filter.fir_filter_ccf(1, firdes.low_pass(
            10, samp_rate, 1e6, 100e3, firdes.WIN_HAMMING, 6.76))
        self.fir_filter_xxx_0 = filter.fir_filter_fff(8, ((0, 1, 0)))
        self.fir_filter_xxx_0.declare_sample_delay(0)
        self.digital_gfsk_demod_0 = digital.gfsk_demod(
            samples_per_symbol=125,
            sensitivity=500e-3,
            gain_mu=0.175,
            mu=0.5,
            omega_relative_limit=0.005,
            freq_error=0.0,
            verbose=False,
            log=False,
        )
        self.blocks_stream_to_vector_1 = blocks.stream_to_vector(
        gr.sizeof_float*1, 1024)
        self.blocks_multiply_xx_0 = blocks.multiply_vcc(1)
        self.blocks_char_to_float_0 = blocks.char_to_float(1, 1)
        self.analog_simple_squelch_cc_0 = analog.simple_squelch_cc(-50, 500e-3)
        self.analog_sig_source_x_0 = analog.sig_source_c(samp_rate,
        analog.GR_COS_WAVE, -3e6, 1, 0)

        ##################################################
        # Connections
        ##################################################(-
        self.connect((self.analog_sig_source_x_0, 0),
        (self.blocks_multiply_xx_0, 1))
        self.connect((self.analog_simple_squelch_cc_0, 0),
        (self.digital_gfsk_demod_0, 0))
        self.connect((self.blocks_char_to_float_0, 0), (self.fir_filter_xxx_0, 0))
        self.connect((self.blocks_multiply_xx_0, 0), (self.low_pass_filter_0, 0))
        self.connect((self.blocks_stream_to_vector_1, 0), (self.data_probe, 0))
        self.connect((self.digital_gfsk_demod_0, 0),
        (self.blocks_char_to_float_0, 0))
        self.connect((self.fir_filter_xxx_0, 0),
        (self.blocks_stream_to_vector_1, 0))
        self.connect((self.low_pass_filter_0, 0),
        (self.analog_simple_squelch_cc_0, 0))
        self.connect((self.osmosdr_source_0, 0), (self.blocks_multiply_xx_0, 0))

    def get_variable_function_probe_1(self):
        return self.variable_function_probe_1

    def set_variable_function_probe_1(self, variable_function_probe_1):
        self.variable_function_probe_1 = variable_function_probe_1

    def get_samp_rate(self):
        return self.samp_rate

    def set_samp_rate(self, samp_rate):
        self.samp_rate = samp_rate
        self.analog_sig_source_x_0.set_sampling_freq(self.samp_rate)
        self.low_pass_filter_0.set_taps(firdes.low_pass(10, self.samp_rate, 1e6,
        100e3, firdes.WIN_HAMMING, 6.76))
        self.osmosdr_source_0.set_sample_rate(self.samp_rate)

    def get_data(self):
    vals = self.data_probe.level()
    return vals

    #Converts list of integers representing binary number (ex: [1,0,1,0]) to
    #integer (ex: 10)
    def frombit_toint(self, ls):
    length = len(ls)
    num = 0
    for i in range(length, 0, -1):
        num += ls[i-1] * 2 ** (length - i)
    return num

    #Convert list of integers representing binary number (ex: [1,0,1,0]) to
    #bit string
    def frombit_tobyte(self, ls):
    return bin(self.frombit_toint(ls))

    #Calculate checksum (XOR)
    def checkcalc(self, key, payload):
    return key ^ self.frombit_toint(payload)


def main():

    def fromint_tobit(num):
        bit = [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
        for i in range(15, -1,-1):
            bit[i] = (num - (num % (2**i))) / (2**i)
            num = num % (2**i)
        bit.reverse()
        return bit

    chrp_tx = chirp_tx()
    chrp_tx.start()

    s_time = time.time()

    FIFO = '/tmp/sddvna.fifo'

    fft_bins = 128*2
    fft_reso = (chrp_tx.samp_rate / fft_bins)


    last_probe = 0
    l_time = time.time()
    tot = 0
    cnt = 0
    chrp_tx.set_ampl(0)
    time.sleep(5)
    first = True
    if chrp_tx.start_freq != 2.475e9:
        for i in range((fft_bins/4),((3*fft_bins/4))):
        chrp_tx.set_freq(i*fft_reso - chrp_tx.samp_rate/2)
        if first:
            chrp_tx.set_ampl(chrp_tx.req_ampl)
            time.sleep(.1)
            first = False

        print("i: %d" % (i))
        print("TX Freq: %.3f" % (chrp_tx.get_freq() + chrp_tx.start_freq))

        waiting = True
        start = time.time()
        while waiting:
            if time.time() - start > .1:
                waiting = False
        cnt += 1
    else:
        for i in range((fft_bins/4),((3*fft_bins/4)+1)):
        chrp_tx.set_freq(i*fft_reso - chrp_tx.samp_rate/2)
        if first:
            chrp_tx.set_ampl(chrp_tx.req_ampl)
            time.sleep(.1)
            first = False

        print("i: %d" % (i))
        print("TX Freq: %.3f" % (chrp_tx.get_freq() + chrp_tx.start_freq))

        waiting = True
        start = time.time()
        while waiting:
            if time.time() - start > .1:
                waiting = False
        cnt += 1


    print('tones transmitted: %d' % cnt)
    time.sleep(1)



    '''     FIFO Stuff
    waiting = True
    if chrp_tx.end == False:
            print('waiting for ack')
        while waiting:
        try:
            os.mkfifo(FIFO)
        except OSError:
            pass
        try:
            fd = os.open(FIFO, os.O_RDONLY|os.O_NONBLOCK)
            fifo_reader = os.fdopen(fd, 'r')
            pipe = fifo_reader.read()
            if pipe == 'Done':
                waiting = False
                print('Pipe says done')
                os.unlink(FIFO)
        except IOError:
            waiting = True

    '''
    hand_rx = handshake_RX()
    hand_rx.start()

    preamble = [1,0,1,0,1,0,1,0]
    checksum_key = 0xFAFA

    #Wait to receive ack
    ack = False
    ack1 = False
    ack2 = False
    ack3 = False
    while ack == False:
    #Get captured data
    data = map(int, list(hand_rx.get_data()))
    #Look for preamble in captured data (0xAA)
    for i in range(0, len(data)):
        if data[i: i+8] == preamble:
            #Get length of payload
            length = hand_rx.frombit_toint(data[i+8: i+16])
            #Get payload
            payload = data[i+16: i+16+length]
            #Compare transmitted and calculated checksums
            check_TX = hand_rx.frombit_toint(data[i+16+length: i+32+length])
            check_calc = hand_rx.checkcalc(checksum_key, payload)
            #If checksums match, then packet received
            if check_TX == check_calc:
                #Wait for ACK from every radio in network
                if payload == fromint_tobit(int("0x2045", 16)):
                    print('Ack from RX1 received')
                    ack1 = True
                elif payload == fromint_tobit(int("0x0EFC", 16)):
                    print('Ack from RX2 Recevied')
                    ack2 = True
                elif payload == fromint_tobit(int("0x1AB4", 16)):
                    print('Ack from RX3 Received')
                    ack3 = True
                if ack1 == True and ack2 == True and ack3 == True:
                    ack = True

    #time.sleep(0.0001)
```

```
    #print('Ack Received!')

    print('Elapsed Time: %.6f' % (time.time() - s_time))

    chrp_tx.set_ampl(0)
    time.sleep(0.1)
    print('Setting signal amplitude to zero')

    hand_rx.stop()
    hand_rx.wait()

    chrp_tx.stop()
    chrp_tx.wait()

    print('Done')
    time.sleep(5)




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

A.2 Receiver

This section includes the code written for the receiving end of the measurement system. Code is included that is used for a single receiver, in a multi-link measurement system.

A.2.1 Shell Script

```
#!/bin/bash

echo "2.4 GHz ISM Frequency Sweep - Receiver"

for ((i = 0; i < 8; i = i+1))
    do
        python RX_1.py -c 2.4${i}5e9 -n ${i}
        wait
    done

echo "Done"
```

A.2.2 Python Code

```
#!/usr/bin/env python2
# -*- coding: utf-8 -*-
##################################################
# GNU Radio Python Flow Graph
# Title: Chirp Rx
# Generated: Mon May 21 11:19:28 2018
##################################################

from gnuradio import blocks
from gnuradio import digital
from gnuradio import eng_notation
from gnuradio import gr
from gnuradio.eng_option import eng_option
from gnuradio.fft import window
from gnuradio.filter import firdes
from gnuradio import filter
from optparse import OptionParser
import osmosdr
import time, os
import threading
import numpy as np
import datetime

class chirp_rx(gr.top_block):

    def __init__(self):

    # Command line parsing
    parser = OptionParser()
    parser.add_option("-c", "--center", dest="center_freq",
        help="Desired Center Frequency", default=2.405e9)
    parser.add_option("-n", "--number", dest="sweep_number",
        help="The number of the current sweep (1-8)", default=1)
    (options, args) = parser.parse_args()

        gr.top_block.__init__(self, "Chirp Rx")

        ##################################################
        # Variables
        ##################################################
        self.samp_rate = samp_rate = 20e6
        self.center_freq = center_freq = float(options.center_freq)
    self.sweep_num = int(options.sweep_number)

        ##################################################
        # Blocks
        ##################################################
        self.val = blocks.probe_signal_c()      #***
        def _variable_function_probe_0_probe():
            while True:
                val = self.val.level()
                try:
                    self.set_variable_function_probe_0(val)
                except AttributeError:
                    pass
                time.sleep(1.0 / (10))
        _variable_function_probe_0_thread = threading.Thread(target =
        _variable_function_probe_0_probe)
        _variable_function_probe_0_thread.daemon = True
        _variable_function_probe_0_thread.start()

        self.osmosdr_source_0 = osmosdr.source( args="numchan=" + str(1) + " " +
         "bladerf=2045f55f182fbb2316da78d626ef7b31" )
        self.osmosdr_source_0.set_time_now(osmosdr.time_spec_t(time.time()),
        osmosdr.ALL_MBOARDS)
        self.osmosdr_source_0.set_sample_rate(samp_rate)
        self.osmosdr_source_0.set_center_freq(center_freq, 0)
        self.osmosdr_source_0.set_freq_corr(0, 0)
        self.osmosdr_source_0.set_dc_offset_mode(0, 0)
        self.osmosdr_source_0.set_iq_balance_mode(0, 0)
        self.osmosdr_source_0.set_gain_mode(False, 0)
        self.osmosdr_source_0.set_gain(0, 0)
        self.osmosdr_source_0.set_if_gain(0, 0)
        self.osmosdr_source_0.set_bb_gain(0, 0)
        self.osmosdr_source_0.set_antenna("", 0)
        self.osmosdr_source_0.set_bandwidth(0, 0)

    file_loc = "/media/jjamison/LaCie/Lab/sweep_data/link1_data/test_%s.bin" \
        % self.sweep_num
        self.blocks_file_sink_0 = blocks.file_sink(gr.sizeof_gr_complex*1,
        file_loc, False)
        self.blocks_file_sink_0.set_unbuffered(False)

    ##
    self.dc_blocker_xx_0 = filter.dc_blocker_cc(10, True)

        ##################################################
        # Connections
        ##################################################
    self.connect((self.osmosdr_source_0, 0), (self.dc_blocker_xx_0, 0))
        self.connect((self.osmosdr_source_0, 0), (self.blocks_file_sink_0, 0))
    self.connect((self.dc_blocker_xx_0, 0), (self.val, 0))  #***

    def get_pow(self):      #***
```

```
            mag = np.absolute(self.variable_function_probe_0)
        if mag != 0:
            return 20*np.log10(mag)
        else:
            return 0

    def set_variable_function_probe_0(self, variable_function_probe_0):     #***
        self.variable_function_probe_0 = variable_function_probe_0

    def get_samp_rate(self):
        return self.samp_rate

    def set_samp_rate(self, samp_rate):
        self.samp_rate = samp_rate
        self.osmosdr_source_0.set_sample_rate(self.samp_rate)

    def get_center_freq(self):
        return self.center_freq

    def set_center_freq(self, center_freq):
        self.center_freq = center_freq
        self.osmosdr_source_0.set_center_freq(self.center_freq, 0)


class handshake_TX(gr.top_block):

    def __init__(self):

    #Converts list of integers representing binary number (ex: [1,0,1,0])
    #    to integer (ex: 10)
    def frombit_toint(ls):
        length = len(ls)
        num = 0
        for i in range(length, 0, -1):
            num += ls[i-1] * 2 ** (length - i)
        return num
        #Calculate checksum (XOR)
    def checkcalc(key, payload):
        return key ^ frombit_toint(payload)

    #Convert number to list of integers representing binary number
    def fromint_tobit(num):
        bit = [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
        for i in range(15, -1,-1):
            bit[i] = (num - (num % (2**i))) / (2**i)
            num = num % (2**i)
        bit.reverse()
        return bit

        gr.top_block.__init__(self, "Handshake Tx")

        ##################################################
        # Variables
        ##################################################
        self.samp_rate = samp_rate = 10e6
        self.samp_per_sym = samp_per_sym = 125
        self.fsk_deviation_hz = fsk_deviation_hz = 500e-3

        ##################################################
        # Blocks
        ##################################################
        self.osmosdr_sink_0 = osmosdr.sink( args="numchan=" + str(1) + " " +
        "bladerf=2045f55f182fbb2316da78d626ef7b31" )
        self.osmosdr_sink_0.set_time_now(osmosdr.time_spec_t(time.time()),
         osmosdr.ALL_MBOARDS)
        self.osmosdr_sink_0.set_sample_rate(samp_rate)
        self.osmosdr_sink_0.set_center_freq(305e6, 0)
        self.osmosdr_sink_0.set_freq_corr(0, 0)
        self.osmosdr_sink_0.set_gain(20, 0)
        self.osmosdr_sink_0.set_if_gain(0, 0)
        self.osmosdr_sink_0.set_bb_gain(0, 0)
        self.osmosdr_sink_0.set_antenna("", 0)
        self.osmosdr_sink_0.set_bandwidth(0, 0)

        self.digital_gfsk_mod_0 = digital.gfsk_mod(
            samples_per_symbol=samp_per_sym,
            sensitivity=fsk_deviation_hz,
            bt=0.35,
            verbose=True,
            log=False,
        )

    preamble = (1,0,1,0,1,0,1,0)
    length = (0,0,0,1,0,0,0,0)
    payload = tuple(fromint_tobit(int("0x2045", 16)))
    #checksum = (0,0,0,0,1,0,0,1,1,1,1,1,0,1,1,0)
    key = 0xFAFA
    checksum = tuple(fromint_tobit(checkcalc(key, payload)))
        self.blocks_vector_source_x_0 = blocks.vector_source_b(
        tuple(preamble+length+payload+checksum), True, 1, [])

        ##################################################
        # Connections
        ##################################################
        self.connect((self.blocks_vector_source_x_0, 0),
        (self.digital_gfsk_mod_0, 0))
```

```
            self.connect((self.digital_gfsk_mod_0, 0),
            (self.osmosdr_sink_0, 0))

    def get_samp_rate(self):
        return self.samp_rate

    def set_samp_rate(self, samp_rate):
        self.samp_rate = samp_rate
        self.osmosdr_sink_0.set_sample_rate(self.samp_rate)

    def get_samp_per_sym(self):
        return self.samp_per_sym

    def set_samp_per_sym(self, samp_per_sym):
        self.samp_per_sym = samp_per_sym

    def get_fsk_deviation_hz(self):
        return self.fsk_deviation_hz

    def set_fsk_deviation_hz(self, fsk_deviation_hz):
        self.fsk_deviation_hz = fsk_deviation_hz

    def set_transmit_freq(self, transmit_freq):
    self.transmist_freq = transmit_freq
    self.osmosdr_sink_0.set_center_freq(transmit_freq, 0)


def main():

    chrp_rx = chirp_rx()
    chrp_rx.start()

    print('Connected to radio')

    print('Center Frequency: %.2f' % chrp_rx.get_center_freq())
    #FIFO = '/tmp/sddvna.fifo'

    meas = True
    powers = []

    noise_floor = None

    sweeping = []

    i = 0

    sweep_start = False

```

```
start_time = None
print('Saving IQ data...')
while meas:
if start_time == None:
    #print('here')
    cur_pow = chrp_rx.get_pow()
    if cur_pow != 0:
        powers.append(cur_pow)
        if noise_floor != None:
            i += 1
            if cur_pow - noise_floor > 15:
                if i > 3000:
                    sweeping.append(True)
                    sweeping.pop(0)
                else:
                    sweeping.append(True)
            else:
                if i > 3000:
                    sweeping.append(False)
                    sweeping.pop(0)
                else:
                    sweeping.append(False)

    if len(powers) > 500:
        noise_floor = np.amax(powers[0:500])
        #print(sum(sweeping))
        if sum(sweeping) > 1500:
            #print('sweeping')
            if sweep_start == False:
                print('Begining of Sweep Detected')
                start_time = time.time()
                sweep_start = True

else:
    if time.time() - start_time >= 16:
        meas = False
        print('Elapsed time since begining of sweep: %.3f\n +
            Ending measurement' % (time.time() - start_time))



hand_tx = handshake_TX()
hand_tx.start()
print('Transmitting ACK')
waiting = True
powers = []
i = 0
under = 0
```

```
    while waiting:
    sec = datetime.datetime.now().second
    if (sec % 3 == 1):
        hand_tx.set_transmit_freq(305e6)
    else:
        hand_tx.set_transmit_freq(355e6)
    cur_pow = chrp_rx.get_pow()
    if i < 10:
        powers.append(cur_pow)
    elif i == 10:
        ref = np.amax(powers)
        print("REF: %.5f" % ref)
    else:
        if ref - cur_pow >= 10:
            under += 1
            print("RX power less than reference")
            if under > 10:
                print("Chirp gone, switching center frequency")
                waiting = False
        else:
            if under > 0:
                print("Underflow detected, reset count")
                under = 0
    i += 1
    print("Received Power: %.3f" % (cur_pow))
    time.sleep(0.05)

    hand_tx.stop()
    hand_tx.wait()

    chrp_rx.stop()
    chrp_rx.wait()





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

A.3 Frequency Response Calculation

This section includes the code written for purposes of calculating the frequency response from the raw sampled data.

A.3.1 Shell Script

```
#!/bin/bash

echo "2.4 GHz ISM Frequency Sweep - Frequency Response Calculator"

for ((i = 0; i < 8; i = i+1))
    do
        echo "Step ${i}"
        python mag_calc_1.py -n ${i}
        wait
    done

echo "Plotting..."

#python plotter.py

echo "Done"
```

A.3.2 Python Code

```
#!/usr/bin/env python2
# -*- coding: utf-8 -*-
##################################################
# GNU Radio Python Flow Graph
# Title: Chirp Phase Calc Fft
# Generated: Tue Apr 24 12:28:14 2018
##################################################

from gnuradio import blocks
from gnuradio import eng_notation
from gnuradio import fft
from gnuradio import gr
from gnuradio.eng_option import eng_option
from gnuradio.fft import window
from gnuradio.filter import firdes
from gnuradio import filter
from optparse import OptionParser
import threading
import time


##  Added imports       ##
import numpy as np
import matplotlib.pyplot as plt
import scipy.signal as sci
```

```


class chirp_phase_calc_fft(gr.top_block):

    def __init__(self):

    # Command line parsing
    parser = OptionParser()
    parser.add_option("-n", "--number", dest="sweep_number",
        help="The number of the current sweep (1-8)", default=1)
    (options, args) = parser.parse_args()
    self.sweep_number = options.sweep_number

        gr.top_block.__init__(self, "Chirp Phase Calc Fft")

        ##################################################
        # Variables
        ##################################################
        self.variable_function_probe_0 = variable_function_probe_0 = 0
        self.samp_rate = samp_rate = 20e6
    self.reso = 128*2

        ##################################################
        # Blocks
        ##################################################
        self.fft = blocks.probe_signal_vc(self.reso)
        def _variable_function_probe_0_probe():
            while True:
                val = self.fft.level()
                try:
                    self.set_variable_function_probe_0(val)
                except AttributeError:
                    pass
                time.sleep(1.0 / (10))
        _variable_function_probe_0_thread = threading.Thread(target=
        _variable_function_probe_0_probe)
        _variable_function_probe_0_thread.daemon = True
        _variable_function_probe_0_thread.start()
        self.fft_vxx_0 = fft.fft_vcc(self.reso, True,
        (window.blackmanharris(self.reso)), False, 1)
        self.blocks_throttle_0 = blocks.throttle(gr.sizeof_gr_complex*1,
        samp_rate,True)
        self.blocks_stream_to_vector_0 = blocks.stream_to_vector(
        gr.sizeof_gr_complex*1, self.reso)

    file_loc = "/media/jjamison/LaCie/Lab/sweep_data/link1_data/test_%s.bin" \
        % options.sweep_number
        self.blocks_file_source_0 = blocks.file_source(gr.sizeof_gr_complex*1,
        file_loc, False)

    ##
    self.dc_blocker_xx_0 = filter.dc_blocker_cc(5000, True)

        ##################################################
        # Connections
        ##################################################
        self.connect((self.blocks_file_source_0, 0),
        (self.blocks_throttle_0, 0))
        self.connect((self.blocks_stream_to_vector_0, 0),
        (self.fft_vxx_0, 0))
        self.connect((self.fft_vxx_0, 0), (self.fft, 0))

    ##
    self.connect((self.blocks_throttle_0, 0), (self.dc_blocker_xx_0, 0))
    self.connect((self.dc_blocker_xx_0, 0), (self.blocks_stream_to_vector_0, 0))

    def get_variable_function_probe_0(self):
        return self.variable_function_probe_0

    def set_variable_function_probe_0(self, variable_function_probe_0):
        self.variable_function_probe_0 = variable_function_probe_0

    def get_samp_rate(self):
        return self.samp_rate

    def set_samp_rate(self, samp_rate):
        self.samp_rate = samp_rate
        self.blocks_throttle_0.set_sample_rate(self.samp_rate)


def main(top_block_cls=chirp_phase_calc_fft, options=None):


    tb = top_block_cls()
    tb.start()
    meas = True
    mags = []
    mag_measurements = np.zeros((tb.reso))
    last_index = 0
    start = False
    sweep_num = int(tb.sweep_number)
    while meas:
    fft = tb.fft.level()
    max_index = np.argmax(np.absolute(fft))
    if (np.absolute(fft[max_index]) > 0.1 and max_index != 192) or \
         (np.absolute(fft[max_index]) > 1 and max_index == 192):
```

```python
        if last_index == max_index:
            mag = np.absolute(fft[max_index])
            mags.append(mag)
            if sweep_num != 7:
                if max_index == ((tb.reso/4)-1) and start:
                    #print(len(phases))
                    if len(mags) >= 50:
                        meas_mag = np.median(mags)
                        mag_measurements[3*(tb.reso/4)-2] = meas_mag
                        print('Freq Index: %d, Bin: %d' % \
                            ((3*(tb.reso/4)-2), last_index))
                        print("Bin #: %d,   Median Magnitude: %.3f" \
                            % (max_index, meas_mag))
                        meas == False
                        break
            else:
                if max_index == ((tb.reso/4)) and start:
                    #print(len(phases))
                    if len(mags) >= 50:
                        meas_mag = np.median(mags)
                        mag_measurements[3*(tb.reso/4)-1] = meas_mag
                        print('Freq Index: %d, Bin: %d' % (3*(tb.reso/4)-1,
                            last_index))
                        print("Bin #: %d,   Median Magnitude: %.3f" % \
                            (max_index, meas_mag))
                        meas == False
                        break
        elif last_index != max_index:
            if last_index == (3*(tb.reso/4)) or start:
                start = True
                if np.array(mags).size >= 50:
                    meas_mag = np.median(mags)
                    val = tb.reso/2 + 1
                    if last_index >= val:
                        freq_step = last_index - val
                    else:
                        freq_step = last_index + tb.reso/2 - 1
                    print('Freq Index: %d, Bin: %d' % (freq_step, last_index))
                    if mag_measurements[freq_step + 1] == 0:
                        print("Bin #: %d,   Median Magnitude: %.3f, "+
                            "Length: %d" % (last_index, meas_mag, len(mags)))
                        mags = []
                        if mag_measurements[freq_step] == 0:
                            mag_measurements[freq_step] = meas_mag
                        else:
                            old_mag_meas = mag_measurements[freq_step]
                            if old_mag_meas - meas_mag > .5:
                                print("***Previous measurement not used***")
                            elif old_mag_meas - meas_mag < -.5:
                                print("***Previous measurement replacing \
                                    existing measurements***")
                                mag_measurements[freq_step] = meas_mag
                            else:
                                mag_measurements[freq_step] = .5 * \
                                    (meas_mag + old_mag_meas)

        last_index = max_index

    #only using inner 10 MHz
    if sweep_num != 7:
        mag_measurements = mag_measurements[(tb.reso/4)-1:3*(tb.reso/4)-1]
    else:
    mag_measurements = mag_measurements[(tb.reso/4)-1:3*(tb.reso/4)]

    #interpolate any missed measurement points
    miss_ind = []
    for j in range(0, len(mag_measurements)):
    if mag_measurements[j] == 0:
        if j == 126:
            mag_measurements[j] = mag_measurements[j-1]
            print("Index %d was missed. Using neighboring indice:" +
                "\nmeas[%d] = meas[%d]" +
                "= %.3f" % (j,j,j-1,mag_measurements[j]))
        elif j == 0:
            mag_measurements[j] = mag_measurements[j+1]
            print("Index %d was missed. Using neighboring indice:" +
                "\nmeas[%d] = meas[%d]" +
                "= %.3f" % (j,j,j+1,mag_measurements[j]))
        elif mag_measurements[j+1] == 0:
            mag_measurements[j] = mag_measurements[j-1]
            print("Index %d and %d were missed.  Using %d indice" +
                " for %d:\nmeas[%d] = %.3f" \
                 % (j,j+1,j-1,j,j,j-1,mag_measurements[j]))
        else:
            mag_measurements[j] = .5 * (mag_measurements[j-1] +
                mag_measurements[j+1])
            print("Index %d was missed.  Averaged neigboring indices:" +
                "\nmeas[%d] = .5 * (meas[%d] + meas[%d]) = .5 * " +
                "(%.3f + %.3f) = %.3f" % (j,j,j-1,j+1, \
                mag_measurements[j-1], mag_measurements[j+1],
                mag_measurements[j]))


    print(len(mag_measurements.tolist()))
    #db = 20*np.log10((1.0/128.0)*np.array(mag_measurements))
    #plt.subplot(2,1,1)
```

```
    #plt.plot(db)
    #plt.subplot(2,1,2)
    #plt.plot(sci.medfilt(db,5))
    #plt.show()
    tb.stop()
    tb.wait()

    test_num = int(tb.sweep_number)
    file_name = '/media/jjamison/LaCie/Lab/sweep_data/link1_data/' +
        'step_%d.txt' % test_num
    f = open(file_name, 'w')
    f.write(str(mag_measurements.tolist()))


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

A.4 Data Analysis Code

This section contains the Python code used to analyze and graph the data captured during the measurement campaign in Chapter 5.

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.patches import Circle
import scipy.signal as sci
from optparse import OptionParser
import csv
from scipy.integrate import quad

def file_parser(file_name):
    file_object = open(file_name, "r")
    string = file_object.read()
    data = []
    str_num = ''
    for i in range(0, len(string)):
        if string[i] != '[' and string[i] != ',':
            if not string[i].isspace() and string[i] != ']':
                str_num += string[i]
            else:
                number = float(str_num)
                data.append(number)
                str_num = ''
    data = 20*np.log10((1.0/256.0)*np.array(data))
    return data

```

```python
def cal_parser(cal_file):
    file_object = open(cal_file, "r")
    string = file_object.read()
    data = []
    str_num = ''
    for i in range(0, len(string)):
        if string[i] != '[' and string[i] != ',':
            if not string[i].isspace() and string[i] != ']':
                str_num += string[i]
            else:
                number = float(str_num)
                data.append(number)
                str_num = ''

    return data

def calc_error(sdr, vna):
    sdr_raw = 10**(sdr/20.0)
    vna_raw = 10**(vna/20.0)
    sdr_reso = len(sdr)
    vna_reso = len(vna)
    err = np.zeros(sdr_reso)
    err_raw = np.zeros(sdr_reso)
    mse = 0
    for i in range(0, sdr_reso):
        err[i] = abs(sdr[i] - vna[int((i/float(sdr_reso))*float(vna_reso))])
        err_raw[i] = abs(sdr_raw[i] - vna_raw[int((i/float(sdr_reso))
            *float(vna_reso))])
        mse += abs(sdr_raw[i] - vna_raw[int((i/float(sdr_reso))
            *float(vna_reso))])**2


    med_err = np.median(err)
    old_avg_err = np.mean(err)

    std = np.std(err_raw)
    avg_err_raw = np.mean(err_raw) / std

    mse_std = (mse / sdr_reso) / std

    return med_err, old_avg_err, avg_err_raw, mse_std, mse

def empirical_cdf(data):
    sorted_data = np.sort(data)
    f = np.array(range(len(data)))/float(len(data))
    return sorted_data, f

def raylcdf(xr):
    raypdf = lambda t: t*np.exp((-t**2)/2)
    r_cdf = np.zeros(len(xr))
    for i in range(0, len(xr)):
        ans, err = quad(raypdf, 0, xr[i])
        r_cdf[i] = ans

    k = np.where((r_cdf >= 0.499) & (r_cdf <= 0.501))
    xr_log = 20*np.log10(xr) - 20*np.log10(xr[k])

    return xr_log, r_cdf


def main():


    cal = cal_parser("../7.9/cal/calibration.txt")

    #Caculate Rayleigh CDF
    xr = np.linspace(0.04, 4, 1000)
    x_cdf, r_cdf = raylcdf(xr)
    #print(x_cdf)
    #print(r_cdf)
    index = np.argmin(abs(r_cdf - .1))
    ray_FD = x_cdf[index]
    boundary = 0.5
    print('Rayleigh 10 perc FD: %.5f' % ray_FD)


    #Get SDR data
    sdr_reso = 128*7 + 129
    link1_data = np.zeros((14, sdr_reso))
    link2_data = np.zeros((14, sdr_reso))
    link3_data = np.zeros((14, sdr_reso))
    sdr_freq = np.linspace(2.4, 2.48, sdr_reso)

    cnt = 0

    '''
           ricean   rayleigh   hyp-ray
    Link 1:  |    -    |     -     |    -    |
    Link 2:  |    -    |     -     |    -    |
    Link 3:  |    -    |     -     |    -    |
    '''
    stats = np.zeros((3,3))

    for k in range(0,3):
        plt.figure(k)
        for j in range(0,14):
            data = np.zeros((7, 128))
            for i in range(0,7):
                data[i] = file_parser("chamber/link_%d/pos_%d/step_%d.txt"
                    % (k+1,j+1,i))
            data8 = np.zeros(129)
            data8 = file_parser("chamber/link_%d/pos_%d/step_7.txt"
                    % (k+1,j+1))
            data = sci.medfilt(np.append(data.flatten(), data8),13) - cal
            cdf, f = empirical_cdf(data)
            cdf = cdf - np.median(cdf)
            FD = cdf[int(len(cdf)*.1)]
            fadetype = None
            if FD > (ray_FD + boundary):
                fadetype = 'Ricean'
                stats[k,0] += 1
            elif FD < (ray_FD - boundary):
                fadetype = 'hyper-Rayleigh'
                stats[k,2] += 1
            else:
                fadetype = 'approximately Rayleigh'
                stats[k,1] += 1
            print('Link: %d, \tPosition: %d, \t10 percent FD: %.5f dBm' +
                    ', \tFading Type: %s' % (k+1,j+1,FD, fadetype))
            plt.semilogy(cdf, f)
        plt.semilogy(x_cdf, r_cdf, 'k')
        plt.title('CDF of $S_{%d1}$ Measurements' % (k+2))
        plt.ylim(.002, 1)
        plt.xlim(-35, 10)
        plt.grid()
        plt.xlabel('Received Signal Strength (dBm)')
        plt.ylabel(r'P[$S_{21}$ < abscissa]')

    for k in range(0,3):
        plt.figure(3)
        for j in range(0,14):
            data = np.zeros((7, 128))
            for i in range(0,7):
                data[i] = file_parser("chamber/link_%d/pos_%d/step_%d.txt"
                        % (k+1,j+1,i))
            data8 = np.zeros(129)
            data8 = file_parser("chamber/link_%d/pos_%d/step_7.txt" % (k+1,j+1))
            data = sci.medfilt(np.append(data.flatten(), data8),13) - cal
            cdf, f = empirical_cdf(data)
            cdf = cdf - np.median(cdf)
            FD = cdf[int(len(cdf)*.1)]
            fadetype = None
            if FD > (ray_FD + boundary):
                fadetype = 'Ricean'
                #stats[k,0] += 1
            elif FD < (ray_FD - boundary):
                fadetype = 'hyper-Rayleigh'
                #stats[k,2] += 1
            else:
                fadetype = 'approximately Rayleigh'
                #stats[k,1] += 1
            print('Link: %d, \tPosition: %d, \t10 percent FD: %.5f dBm' +
                    ', \tFading Type: %s' % (k+1,j+1,FD, fadetype))
            plt.semilogy(cdf, f)
        plt.semilogy(x_cdf, r_cdf, 'k')
        plt.title('CDF of All Measurements')
        plt.ylim(.002, 1)
        plt.xlim(-35, 10)
        plt.grid()
        plt.xlabel('Received Signal Strength (dBm)')
        plt.ylabel(r'P[$S_{21}$ < abscissa]')

    #Plot two link3 plots next to each other
    sdr_reso = 128*7 + 129
    sdr_freq = np.linspace(2.4, 2.48, sdr_reso)
    link3_data = np.zeros((2, sdr_reso))
    plt.figure(4)
    for j in range(0,2):
        data = np.zeros((7, 128))
        for i in range(0,7):
            data[i] = file_parser("chamber/link_3/pos_%d/step_%d.txt" % (j+1,i))
        data8 = np.zeros(129)
        data8 = file_parser("chamber/link_3/pos_%d/step_7.txt" % (j+1))
        link3_data[j] = sci.medfilt(np.append(data.flatten(), data8),3) - cal

    #plt.subplot(2,1,1)
    #cdf1, f1 = empirical_cdf(link3_data[0])
    #cdf1 = cdf1 - np.median(cdf1)
    #plt.semilogy(cdf1, f1, 'k')


    #cdf2, f2 = empirical_cdf(link3_data[1])
    #cdf2 = cdf2 - np.median(cdf2)
    #plt.semilogy(cdf2, f2, 'k--')

    #plt.subplot(2,1,2)
    plt.plot(sdr_freq, link3_data[0],'k')
    plt.plot(sdr_freq, link3_data[1],'k--')
    plt.grid(True)
    plt.xlim(2.4, 2.48)
    plt.xlabel('Frequency (GHz)')
    plt.ylabel('Magnitude (dBm)')
```

```
plt.title(r'Effects of Moving $RX_3$ on $S_{41}$')
plt.legend(('Measurement Taken at x=0cm', 'Measurements Taken at x=5cm'))

plt.figure(5)
for j in range(0,2):
    data = np.zeros((7, 128))
    for i in range(0,7):
        data[i] = file_parser("chamber/link_1/pos_%d/step_%d.txt" % (j+1,i))
    data8 = np.zeros(129)
    data8 = file_parser("chamber/link_1/pos_%d/step_7.txt" % (j+1))
    link3_data[j] = sci.medfilt(np.append(data.flatten(), data8),5) - cal

#plt.subplot(2,1,1)
#cdf1, f1 = empirical_cdf(link3_data[0])
#cdf1 = cdf1 - np.median(cdf1)
#plt.semilogy(cdf1, f1, 'k')


#cdf2, f2 = empirical_cdf(link3_data[1])
#cdf2 = cdf2 - np.median(cdf2)
#plt.semilogy(cdf2, f2, 'k--')

#plt.subplot(2,1,2)
plt.plot(sdr_freq, link3_data[0],'k')
plt.plot(sdr_freq, link3_data[1],'k--')
plt.grid(True)
plt.xlim(2.4, 2.48)
plt.xlabel('Frequency (GHz)')
plt.ylabel('Magnitude (dBm)')
plt.title(r'Effects of Moving $RX_3$ on $S_{21}$')
plt.legend(('Measurement Taken at x=0cm', 'Measurements Taken at x=5cm'))

plt.figure(6)
for j in range(0,2):
    data = np.zeros((7, 128))
    for i in range(0,7):
        data[i] = file_parser("chamber/link_2/pos_%d/step_%d.txt" % (j+1,i))
    data8 = np.zeros(129)
    data8 = file_parser("chamber/link_2/pos_%d/step_7.txt" % (j+1))
    link3_data[j] = sci.medfilt(np.append(data.flatten(), data8),5) - cal

#plt.subplot(2,1,1)
#cdf1, f1 = empirical_cdf(link3_data[0])
#cdf1 = cdf1 - np.median(cdf1)
#plt.semilogy(cdf1, f1, 'k')


#cdf2, f2 = empirical_cdf(link3_data[1])
```

```
    #cdf2 = cdf2 - np.median(cdf2)
    #plt.semilogy(cdf2, f2, 'k--')

    #plt.subplot(2,1,2)
    plt.plot(sdr_freq, link3_data[0],'k')
    plt.plot(sdr_freq, link3_data[1],'k--')
    plt.grid(True)
    plt.xlim(2.4, 2.48)
    plt.xlabel('Frequency (GHz)')
    plt.ylabel('Magnitude (dBm)')
    plt.title(r'Effects of Moving $RX_3$ on $S_{31}$')
    plt.legend(('Measurement Taken at x=0cm', 'Measurements Taken at x=5cm'))




    #Summary statistics
    print('\n\t *** Summary of Data *** \n')
    for i in range(0,3):
        print('Link %d --> Ricean: %d\tapproximately Rayleigh: %d\t' +
                'hyper-Rayleigh: %d' % (i, stats[i,0], stats[i,1], stats[i,2]))



    plt.show()




if __name__ == '__main__':

    main()
```