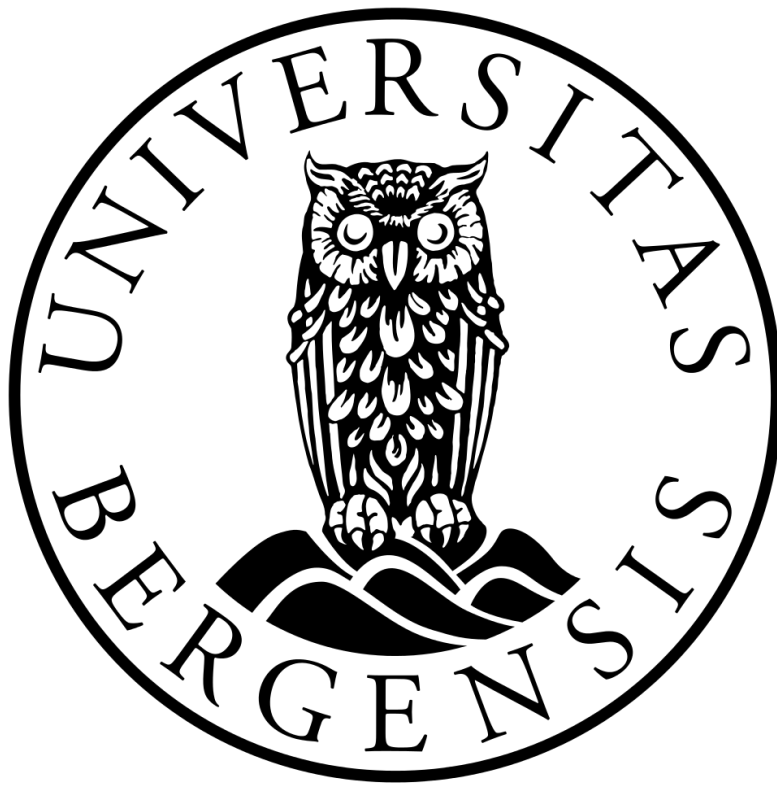




MASTER THESIS

UNIVERSITY OF BERGEN

DEPARTMENT OF INFORMATICS

**Relay-Assisted Wireless Communication:
From Theory to Real-World Implementation
Using Empirical Measurements**

Author:
Hans Trygve Rørlien

Supervisor:
Øyvind Ytrehus

November 20, 2025

Acknowledgements

I would like to start by thanking Øyvind Ytrehus, my supervisor, for attempting to help me stay on track while writing my thesis, and also for the brainstorming sessions, idea sparring and great discussions back and forth.

I would also like to thank Bjørn Møller Greve for brainstorming and providing the initial inspiration for some future hobby projects.

Thanks to the inhabitants of Voss (mostly sheep) for letting me walk around with my antenna and not asking any questions regarding why.

Hans Trygve Rørlien
June 10, 2026

Abstract

The ever increasing demand for high-throughput wireless communication fuels the development of next-generation wireless systems. This thesis investigates methods for improving wireless systems by analyzing the capacity of cooperative relaying schemes using empirical data captured by a spectrum analyzer. The work involves developing code for visualization using theoretical capacity formulas, and lays the groundwork for making realistic simulations that are easily convertible to physical implementations.

The thesis introduces known relaying schemes such as the Amplify-Forward, Decode-Forward, Compress-Forward, and Quantize-Map-Forward. These schemes are used throughout the thesis as examples, more specifically the much-researched single-relay network. By using the empirical data on the established capacity calculations for the single-relay network a visualization for planning node placement is developed. Finally, a guide-like introduction to GNU Radio is created, with a focus on incorporating the empirical data into simulation, as well as a conversion from simulation to a real-world physical implementation using software-defined radios.

After presenting my measured results I find limitations in using a spectrum analyzer to capture the complex effects of wireless communication. I find discrepancies between theoretical results of throughput and real-world achievable throughput, and I present my groundwork for simulating and implementing networks in the signal processing software GNU Radio.

Contents

Acknowledgements	i
Abstract	iii
Contents	v
List of Figures	viii
List of Tables	xi
1 Introduction	1
1.1 Motivation	2
1.2 Objective	3
1.3 Thesis Organization	3
2 Background	4
2.1 Information Theory	5
2.1.1 Relaying	5
2.1.2 Shannon–Hartley Theorem	5
2.1.3 Max-Flow Min-Cut Theorem	5
2.1.4 Cut-Set Upper Bound	6
2.1.5 Amplify-Forward	7
2.1.6 Decode-Forward	7
2.1.7 Compress-Forward	7
2.1.8 Quantize-Map-Forward	8
2.2 Spectrum Analysis and Wireless Measurement	8
2.2.1 Spectrum Analyzer	9
2.2.2 WiFi Adapter	12
2.2.3 Antennas	12
2.3 Effects on Wireless Channels	13
2.3.1 Noise	13
2.3.2 Timing	13
2.3.3 Fading	13
2.3.4 Path Loss	14
2.4 Simulation and Implementation Tools	14
2.4.1 Software-Defined Radio	14
2.4.2 GNU Radio	15
2.4.3 GNU Radio Companion	15
2.5 Related Work	16
3 Methodology	17
3.1 Data Collection	17
3.1.1 Spectrum Analyzer	17

3.1.2	WiFi Adapter	20
3.1.3	Antenna	21
3.1.4	Real-World Measurements	22
3.1.5	Testbed Data	25
3.1.6	Simulated Data	26
3.1.7	Normalizing the data	26
3.2	Placement Based Relay Channel Capacity	27
3.3	Simulation and Implementation in GNU Radio	27
3.3.1	Custom Blocks	28
3.3.2	Flowgraphs and In-Tree Module Blocks	29
3.3.3	Single-Relay Channel Flowgraph	32
3.3.4	Software-Defined Radio Implementation	32
4	Results	34
4.1	Data Findings	34
4.1.1	Measured data	34
4.1.2	Simulated and Testbed Data	36
4.2	Relay Channel Capacity Visualization	37
4.2.1	Real-World Heatmap	39
4.2.2	Scheme Specific Heatmaps	40
4.3	Simulation and Implementation in GNU Radio	41
5	Discussion, Further Work and Conclusion	43
5.1	Discussion and Further work	43
5.1.1	Data Collection	43
5.1.2	Relay Capacity Visualization	44
5.1.3	GNU Radio and Software-Defined Radios	44
5.2	Conclusion	45
	References	45

List of Figures

2.1	A simple three node network showing all cuts separating the Source from the Destination.	6
2.2	A simple three node wireless relay network.	6
2.3	Spectrum analyzer showing a max hold trace on the 2.4 GHz ISM band, capturing a single network using WiFi channel 11.	9
2.4	Spectrum analyzer displaying signal strength (green), noise floor (red), and calculated Signal-to-Noise Ratio (cyan) for a 2.4 GHz signal. . . .	11
2.5	A single-plane simplification of radiation patterns for antennas. . . .	12
3.1	A measured WiFi signal where some parts of the total bandwidth have not been measured yet.	20
3.2	Output of my WiFi scan Python script.	21
3.3	Output of airodump-ng (requires monitor mode).	21
3.4	A Chinese parabolic WiFi antenna.	22
3.5	TP-link router in window.	23
3.6	TinySA screen showing a 20 MHz signal measured from the location of Figure 3.7	24
3.7	Measurement point: 7 meters from the router.	24
3.8	Measurement point: 50 meters from the router	25
3.9	A simplistic GNU Radio simulation visualized in GNU Radio companion.	29
3.10	A single wireless channel with noise in GNU Radio companion.	30
3.11	A single wireless channel with noise, fading and path loss in GNU Radio companion.	32
3.12	Single-relay channel with noise, fading and path loss in GNU Radio companion.	32
3.13	Flowgraph of SDR implementation for source, destination and relay nodes.	33
3.14	Flowgraph of SDR implementation for a source node with GNU Radio modulation and pulse shaping.	33
4.1	A noise floor reading far away from any noise sources.	34
4.2	Screenshots from TinySA measuring our routers signal from 5 distances.	35
4.3	Map of SNR (dB) for each measurement location. Crossed circle (red) is the router, delta (teal) is a measuring location. Google, n.d. [1]	36
4.4	Fitted path loss model vs measured data points in SNR.	38
4.5	Cut-set upper bound capacity heatmap for different relay placements, with source node (teal) on the left and destination node (green) on the right.	39

4.6	Capacity heatmap on top of a real world map. Source node in teal, and destination node in green.	40
4.7	Bits/s/Hz capacity heatmaps for different relay schemes. Source node (teal) is on the left; destination node (green) is on the right.	40
4.8	Single-relay channel with noise, fading and path loss in GNU Radio companion.	41
4.9	Flowgraph of SDR implementation for a source relay node.	41
4.10	Flowgraph of SDR implementation for a destination node.	41
4.11	Flowgraph of SDR implementation for a relay node.	42
4.12	Flowgraph of SDR implementation for a source node with GNU Radio modulation and pulse shaping.	42

List of Tables

3.1	IEEE 802.11 b/g/n 2.4 GHz WiFi Channels and Center Frequencies .	19
3.2	Testbed performance Comparison of Cooperative Relaying Schemes .	26
3.3	Simulated performance of QMF using 16 QAM modulation	26
3.4	Experimental performance of QMF in scenario 1 with different noise floors	27
4.1	Measured Wireless Signal Parameters and Calculated Theoretical Capacities.	36
4.2	Simulated performance of QMF using 16 QAM modulation	37
4.3	Experimental performance of QMF given different noise floors	37
4.4	Throughput (bps/Hz) for simulated quantize-map-forward and theoretical calculations for different schemes	37

Nomenclature

α	Alpha
$\mathcal{B}$	Bandwidth
$\mathcal{C}$	Channel Capacity
$\mathbb{K}$	Set of all cuts separating source and destination
Δ	Delta
ϵ	Epsilon
$\mathbb{F}$	Set of flow values for all cuts
$\mathcal{F}$	Flow value across a given cut
$\mathcal{N}$	Noise Power
Ω	Omega (ohm)
σ	Sigma
$\mathcal{S}$	Signal Power
S	Signal-to-Noise ratio (dB)
A	Linear Amplitude Gain
d	Distance
G	Linear Power Gain
P	Power
PL	Path Loss
Rx	Receiver
Tx	Transmitter

Chapter 1

Introduction

Since the seminal work of Shannon in 1948 the development of digital communication networks has boomed. Starting off most communication was done using electrical signals through wires, however the low cost and convenience of wireless communication makes it an increasingly larger part of communication channels. While there are adverse effects on a signal sent through a wire, these are far from the plethora of effects on wireless signals. Different wireless signals do not necessarily play fair and with the increasing amount of devices the electromagnetic spectrum is ever so crowded. This requires good solutions for sharing the limited space of the radio spectrum, and several tried and tested protocols are used. Most notably the communication standards for WiFi and Bluetooth. These are constantly under development and improvements to meet the demand of not only a more crowded RF space, but also a demand for higher throughput. Developing new technologies is not only about increasing the throughput for a specific link, but comes with several additional benefits. This can range from relieving transmitters from inefficient power usage, increasing the total capacity of the radio spectrum, and improve the effect of covert communication.

One promising method of increasing the quality of a wireless channel is the addition of a third node, a relay. Instead of the direct channel between a source and destination we now have another path for information to travel, and in contrast to wired channels where a signal is limited by the wire, wireless signals are implicitly broadcast. The relays job ranges from retransmitting the whole received signal to transmit side information to help the destination make sense of a received signal. This type of communication channel is called a cooperative relay channel. The three core cooperative relaying schemes are amplify-forward, decode-forward and compress-forward. In this paper I also include an additional method, the quantize-map-forward, which is closely related to compress-forward. The three core relaying schemes were established a long time ago and much work has been done in creating theoretical models for the capacity bound of each scheme.

While the development of new techniques are at their core based off of the earlier theoretical works of Shannon. The complexity of the physics and quirks of wireless communication often calls for simulations or testbed implementations. In the literature we can find several examples of theories being simulated to confirm theories, but these simulations often only take into account noise and simple modulation, while

discarding several adverse effects such as multi-path propagation, synchronization, and also ignoring the negatives of implementing things like superimposing signals, co-existence, overhead and strict latency requirements.

The planning of communication systems require empirical data in which we can use to estimate a real-world usable throughput. We can use the established theoretical works to estimate this for a cooperative relay channel, but they are often far from the usable throughput capacity of a channel. Using the aforementioned simplified simulations will also give us estimates far from realistic throughput.

This study gathers data using a spectrum analyzer which is later used to analyze the different relaying schemes, and I discuss the discrepancies of making simple measurements using a spectrum analyzer. I find methods for planning the placement of a relay node using the known relaying schemes as examples. The thesis also shows how to incorporate the empirical data into GNU Radio simulations. Finally, I show that by making simulations within GNU Radio we can easily convert these to physical implementation using software-defined radios.

1.1 Motivation

In the realm of digital communication, there is an ever increasing demand for wireless communication. Not only is the number of wireless devices increasing but also the demand for more throughput per device. This places a strain on the limited radio frequency spectrum, and the accurate planning and implementation of next-generation wireless systems is paramount. Improving the quality of wireless channel links increases throughput, reduces inefficient power usage at the nodes, enhances general capacity of the wireless spectrum and can improve the effect of covert communication.

Current methods of increasing throughput rely heavily on theoretical models. While this approach is undoubtedly a necessary step of the process, these models often lack the complex and dynamic characteristics of real-world environments. Creating simulations for these models is often a step in the right direction towards realistic estimates. While these simulated models often add some impairments like modulation and noise, they tend to leave out a huge part of the adverse effects and quirks of transmitting over wireless channels.

Through the use of software design for signal processing and real-world implementation like GNU Radio, we can confidently make realistic models by incorporating empirical data. In addition to this we gain the advantage that our simulated model can be easily converted to a physical implemented model using software-defined radios.

By showing the process of measuring signals and incorporating these into realistic GNU Radio simulations, I hope to make the process of creating realistic simulations easier. Furthermore, by using the custom blocks of GNU Radio we can implement any coding technique efficiently using the programming language C++, which can be used for both simulation and implementation. By creating example blocks for

both, I hope to help researchers implement their own coding schemes within them. Hopefully, this bridges the gap between simulation and implementation, making the jump from one to the other easily achievable in a graphical user interface.

1.2 Objective

The primary objective of this thesis is to improve the accessibility of planning, simulating and real-world testing for wireless communication networks such as the single-relay model. Ultimately this work aims to make it easier to create better and more realistic estimates for new wireless communication systems.

- Introduce tools and methods used for measuring wireless signals and improvements to such methods.
- Introduce methods for planning node placement for wireless relay networks.
- Introduce improved methods for simulating wireless communication.
- Show ways to bring simulations to life using software-defined radios and GNU Radio.

The initial goal was to create a working physical implementation to compare throughput results, but this proved to be a bit ambitious for the time frame. Instead, I have done the groundwork for such a task.

1.3 Thesis Organization

The thesis is organized as follows:

Chapter 2 - Background provides the necessary theoretical and practical preliminaries for the simulation, implementation, measurement, and calculation of wireless channels.

Chapter 3 - Methodology shows the methods and tools used for measuring and collecting data. The chapter explains logic and software used in my code, and the last section works as a guide on how to set up an environment for simulation and implementation.

Chapter 4 - Results will present the data collected, show the visualization code in use and show the resulting flowgraphs and code for simulation and implementation.

Chapter 5 - Discussion, Further Work, and Conclusion evaluates the findings and potential limitations of the research, before speculations on additional parameters for real-world data harvesting, code improvements, and thoughts on a continuation of the GNU Radio flowgraphs. Finally, some conclusions surrounding the objective.

It is assumed that the reader of this thesis has a basic understanding of coding theory and wireless communication protocols.

Chapter 2

Background

The study of wireless communication systems requires a comprehensive foundation, spanning fundamental theory, practical protocols, hardware, and experimental platforms.

I begin by introducing the concept of relaying, followed by foundational concepts on information capacity calculations, which are built on Shannon's seminal work on channel capacity, the maximum information rate of a noisy communication channel (see Section 2.1.2). I follow this up with the max-flow min-cut theorem and cut-set upper bound, which establishes the maximum achievable information rate in networks.

These theoretical limits inspire the design of specific relaying schemes, which dictate how information should be processed at each node in a network to maximize efficiency. I will introduce the core principles of four specific relaying schemes, amplify-forward (AF), decode-forward (DF), compress-forward (CF), and quantize-map-forward (QMF). For (AF), (DF), and (CF) capacity formulas for the single-relay network can be found, which will be detailed in their respective sections.

To realize and measure these relaying schemes in a real-world environment, I will introduce some wireless hardware and measuring tools. Since I will be measuring WiFi signals I will briefly mention the use of wireless adapters and how to get an overview in a crowded wireless environment. The following is an in-depth explanation of the settings needed to take measurements using a spectrum analyzer. As I introduce the spectrum analyzer, I will simultaneously introduce key channel parameters that will be measured, more specifically noise floor, signal strength, and signal-to-noise ratio. I will also briefly mention wireless antennas and their specifications, as it is not only used when doing these measurements but can be crucial in improving the quality of real-world wireless channels.

Wireless communication is fundamentally limited by some adverse effects. Some of these are introduced here, and examples are given on how to simulate or calculate the implications for the wireless channel.

Finally, the chapter concludes with a description of software-defined radios (SDR) and covers the core concepts of GNU Radio. While the former is a software con-

figurable radio device, the latter is mainly a means of controlling the said device. This flexible combo is the core technology for this work, enabling not only practical implementation but also programmatic simulations of the discussed relay protocols.

2.1 Information Theory

2.1.1 Relaying

Relaying in itself is simply the assistance of a transmission between some source and destination. In typical wired networks, this means a complete retransmission of a message towards a destination. Here, only the recipient of a message can see it, while wireless communication on the other hand, is implicitly broadcast. Wireless communication channels also suffer significantly from several natural effects, most prominently path loss and fading. To combat these effects and to increase the quality of wireless channels, cooperative relaying was first introduced in [2]. This shifts the communication channel from a source-to-destination link to a model where the source-to-destination link still exists, but is assisted by one or more intermediate relay nodes. The most researched model is the single-relay model, which I will use in examples.

2.1.2 Shannon–Hartley Theorem

The Shannon-Hartley theorem introduced in [3] is an application of Shannon’s channel coding theorem to calculate the capacity of a channel subject to additive white gaussian noise. Given a specific bandwidth $\mathcal{B}$, noise level $\mathcal{N}$, and signal strength $\mathcal{S}$. The channel capacity $\mathcal{C}$ in bits per second is given by:

$$\mathcal{C} = \mathcal{B} \log_2 \left(1 + \frac{\mathcal{S}}{\mathcal{N}} \right)$$

The related formula for maximum theoretical spectral efficiency will also be used, which calculates the bits per second per Hz (bits/s/Hz):

$$\frac{\mathcal{C}}{\mathcal{B}} = \log_2 \left(1 + \frac{\mathcal{S}}{\mathcal{N}} \right)$$

This upper-bound channel capacity calculation is on a coded channel with infinite block length and infinite decoding time, and is an ideal upper limit that coding schemes can only approach.

2.1.3 Max-Flow Min-Cut Theorem

The max-flow min-cut theorem from [4] states that the maximum flow from a single source to a single destination in a network is equal to the minimum flow of any cut that separates the source from the destination.

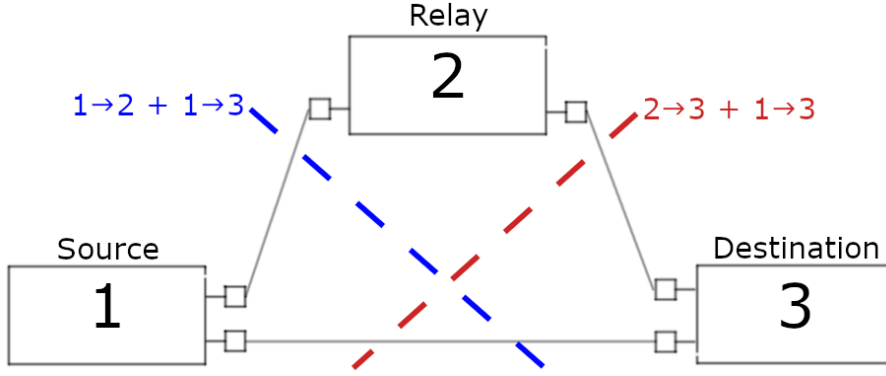


Figure 2.1: A simple three node network showing all cuts separating the Source from the Destination.

We first create a set $\mathbb{K}$ of all possible cuts that separate the source and destination as shown in Figure 2.1. We then calculate the sum flow $\mathcal{F}$ of each cut and create a new set $\mathbb{F} = \{\mathcal{F}_1, \mathcal{F}_2, \dots\}$. The smallest of the $\mathbb{F}$ set is the maximum flow.

2.1.4 Cut-Set Upper Bound

The max-flow min-cut theorem works as an upper bound for networks where each link between nodes are independent. This is not the case for wireless networks, where the same source signal can be received by both the relay and the destination, as shown in Figure 2.2.

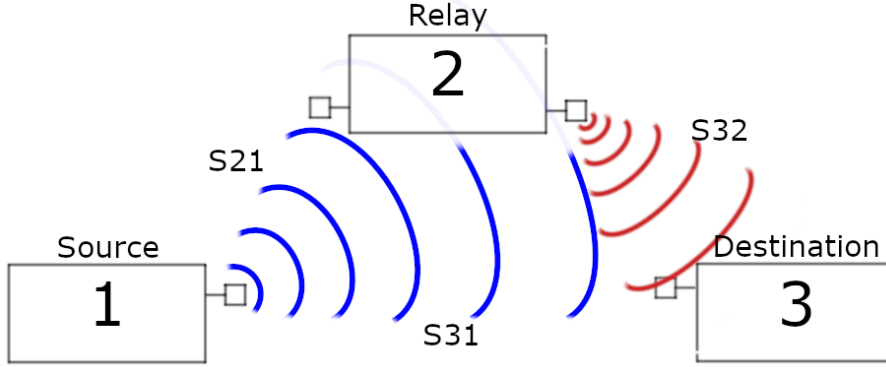


Figure 2.2: A simple three node wireless relay network.

The cut-set upper bound introduced in [2] is instead generalized for communication networks, and considers the mutual information of each cut. From eq. (16.4) in [5] we find a simplified formula for the upper bound of a single relay full-duplex gaussian channel:

$$\begin{cases} \mathcal{C}((\sqrt{S_{21}S_{32}} + \sqrt{S_{31}(S_{31} + S_{21} - S_{32})})^2 / (S_{31} + S_{21})) & \text{if } S_{21} \geq S_{32} \\ \mathcal{C}(S_{31} + S_{21}) & \text{otherwise.} \end{cases}$$

Where S denotes the linear signal-to-noise ratio of each channel, numbered according to Figure 2.2. The capacity function from the Shannon–Hartley theorem in section 2.1.2 is used for $\mathcal{C}(x) = \mathcal{B} \log(1 + x)$.

2.1.5 Amplify-Forward

Amplify-forward (AF) is a facilitation strategy and the most basic and straightforward relaying technique. A relay receives a signal from a source or relay, amplifies it and sends the amplified signal forward to the next relay or source. This technique can be very easy to implement even with simple analog hardware and has the best latency of the schemes, but comes with some inherent limitations. The main drawback comes from the amplification of unwanted signals. In a wireless setting there will always be noise present, and a relay using AF will amplify this noise in addition to the signal. This noise amplification is less prominent in the single-relay network, but for a larger network with several intermediate nodes, the noise can be amplified several times over before reaching its destination.

Using the same terms as the cut-set upper bound in 2.1.4, a formula for the capacity of a simple amplify-forward relaying scheme in eq. (16.18) in [5] is found to be:

$$\mathcal{C}(S_{31} + \frac{S_{21}S_{32}}{S_{21} + S_{32} + 1})$$

2.1.6 Decode-Forward

In the decode-forward (DF) scheme a relay first decodes the received signal, encodes and re-transmits an amplified version to the next relay or receiver. In contrast to the amplify-and-forward scheme the relays in this scheme remove received noise completely before forwarding. Relays can no longer be simple analog amplifiers like in the amplify-forward scheme, and every relay must include some means to decode and encode, which for newer coding schemes is far from trivial. Instead of the noise propagation problem in AF, we get error propagation, where a single error in decoding at any relay will be propagated throughout the network.

The most simple mitigation to the error propagation problem is ensuring good enough link quality for no decoding errors at relays, or for larger networks to selectively send packets through links where the quality is good enough. Techniques using cyclic redundancy checks have also been implemented to stop relays from forwarding packets that are likely to have decoding errors. [6] [7]

From eq. (16.6) in [5] we find a lower-bound for the capacity of a single-relay channel to be:

$$\begin{cases} \mathcal{C}((\sqrt{S_{31}(S_{21} - S_{32})} + \sqrt{S_{32}(S_{21} - S_{31})})^2/S_{21}) & \text{if } S_{21} \geq S_{31} + S_{32} \\ \mathcal{C}(S_{21}) & \text{otherwise.} \end{cases}$$

2.1.7 Compress-Forward

In the compress-forward (CF) technique a relay receives a noisy signal and does a type of lossy compression on the signal before forwarding. More specifically it receives a noisy signal, quantizes it to some discrete representation, and encodes the discrete signal to a shorter codeword. Although some of the information is lost at the relay, it can be used as side information when decoding at the destination.

With the compress-forward scheme we no longer have the problem of noise or error propagation, but instead we have a very high complexity at both the relay and destination. CF also suffers when used in low-latency implementations with shorter block lengths. These latency requirements often found in real-world wireless systems can push compress-forwards achievable rate far below its theoretical limits.

From eq. (16.12) in [5] we find a lower-bound for the capacity of a single-relay to be:

$$\mathcal{C}(S_{31} + \frac{S_{21}S_{32}}{S_{31} + S_{21} + S_{32} + 1})$$

From the equation we can clearly see that compress-forward is guaranteed to achieve a rate that is at least equal to the rate of the direct link from source to destination, and can also in some cases get very close to the cut-set upper bound.

2.1.8 Quantize-Map-Forward

The quantize-map-forward (QMF) scheme is introduced in [8] and captures the core philosophy of CF, but in contrast will achieve the cut-set upper bound within a gap, independent of the channel parameters. They create a deterministic channel model for several of the classic relay setups, including the single-relay model. The goal of this deterministic model is to capture key properties of signal strength, broadcast, and superposition, giving the insight to arrive at the QMF scheme. The scheme replaces the lossy compression of CF with a simpler fixed-alphabet, reducing the complexity of the relays operation. The relay first quantizes the signal to a discrete representation, then maps it to a codeword. This mapping function is typically much simpler than the compression required by CF.

2.2 Spectrum Analysis and Wireless Measurement

In the world of WiFi we use the word channels to describe specific frequency ranges within the WiFi spectrum, and a common analogy is to think of them as different lanes on a highway. While WiFi hardware is forced to use the designated lanes, a spectrum analyzer has a much broader range for choosing both the location and width of its lanes.

Frequency is the rate of oscillation (in Hertz) of a signal, and in wireless communication we often use a range of these. Standard 2.4 GHz WiFi for example uses the frequency span from 2.4 GHz to 2.4835 GHz, and this range is often called a spectrum (the highway). This frequency range is then split into 14 channels (lanes), with each channel totaling a 20 MHz bandwidth (some WiFi protocols use 40 MHz).

Bandwidth can sometimes also be used as the upload/download speed of an internet connection, and most likely has its roots from the fact that the capacity of a channel is proportional to the size of the bandwidth (see 2.1.2). In this thesis I always mean some frequency span when using the word bandwidth.

2.2.1 Spectrum Analyzer

A spectrum analyzer is a tool that displays the magnitude of an input signal as a function of frequency. It typically displays a specified frequency bandwidth on the horizontal axis and a magnitude or power on the vertical axis. A spectrum analyzer is used for several tasks such as wireless network optimization, finding interference sources and can even be used for finding distortions in audio circuits. For our purposes, we are going to use it to measure noise floor and signal strength.

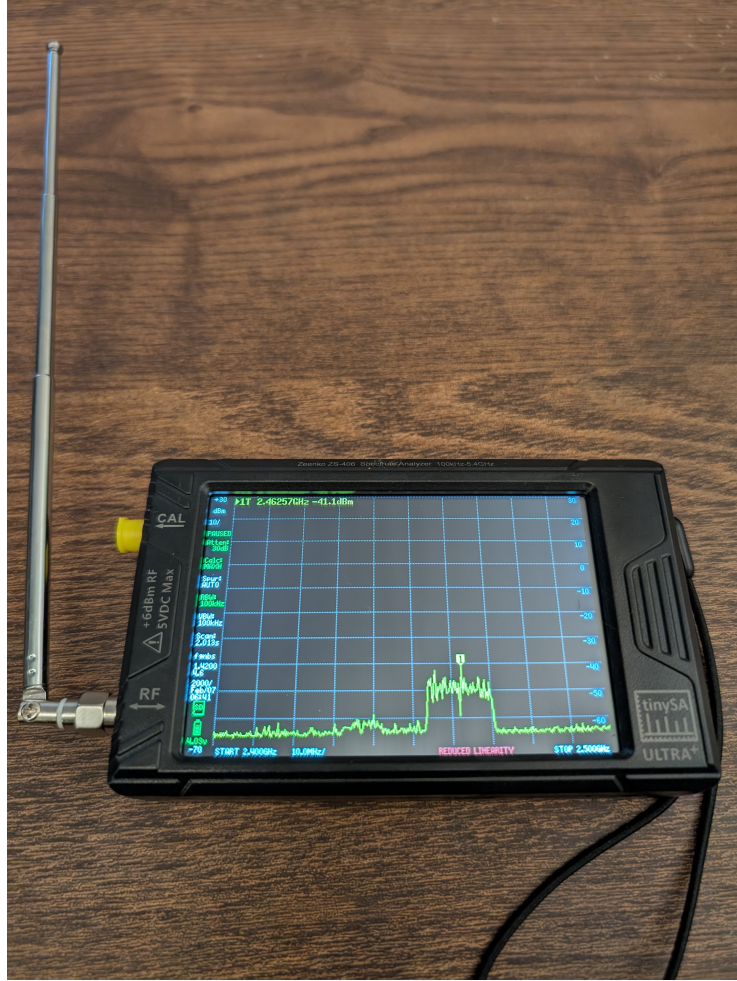


Figure 2.3: Spectrum analyzer showing a max hold trace on the 2.4 GHz ISM band, capturing a single network using WiFi channel 11.

Spectrum analyzers come in different shapes and sizes (and price tags!), but I believe configurations and parameters presented here should work for most analyzers.

Frequency Settings

On the spectrum analyzer we are free to choose the location and size of our bandwidth within the spectrum analyzers specifications. A spectrum analyzer has a very large total frequency range, of which it can tune to read a small part of at any time, often called the acquisition bandwidth. Although the spectrum analyzer can listen to the whole acquisition bandwidth simultaneously, we usually want to limit the listening bandwidth close to our designated signal. We do this by either setting a

start and stop frequency, or we can set the center frequency and a bandwidth size, often referred to as span. Using either method is a choice of convenience.

Resolution Bandwidth

As the spectrum analyzer scans our chosen frequency bandwidth, it has another much narrower filter for internally measuring a specific span of frequencies at any time. The resolution bandwidth (RBW) is the width of this filter and determines the smallest frequency difference the analyzer can distinguish. If the RBW is chosen too wide, it will merge two closely spaced signals together, and more importantly merges noise signals together and raises the displayed noise floor.

Video Bandwidth

The Video Bandwidth (VBW) smooths the displayed trace by averaging the power measurements after the signal has passed through the resolution bandwidth filter. A lower VBW gives more smoothing and noise reduction, which averages our displayed noise floor and can yield more stable measurements.

Attenuation

While the transmitting side of radio frequency (RF) equipment often has a high power usage and output, the receiving side must pick up very low power signals and is very sensitive. Attenuators reduce the strength of an incoming signal to a level the receiver can handle without getting damaged. Attenuator level is measured in decibels (dB); a 10 dB attenuator for example reduces the signal by a factor of 10. On measuring devices we can usually change the displayed offset of the signal to get a correct reading. The offset is simply set to the negative of the total attenuation of external components, and an attenuator of 10 dB requires an offset of -10 dB to get a correct reading.

Noise Floor

Regardless of the equipment used, there will always be some background noise along with a received signal. This noise is a continuous low level background electromagnetic energy present in the environment. These signals are mainly from background radiation, internal noise from the measuring device, and other unwanted signals from electric and electronic equipment.

Signal Strength

Signal strength is the amplitude or power level of our usable signal. A high signal strength is beneficial and means that it is less likely to be corrupted by noise or interference. On the spectrum analyzer, the signal strength can be read from the vertical peak at a specific frequency.

Signal-to-Noise Ratio

The signal-to-noise ratio (SNR), or sometimes signal-to-interference-plus-noise (SINR), is used to determine the reliability of a connection, and the modulation and coding

schemes used are often changed dynamically with a changing SNR (or SINR) value, a process known as dynamic rate switching or rate adaptation.

As the name suggests the SNR value is calculated using the formula:

$$\frac{S}{N} = \frac{\text{Signal Strength}}{\text{Noise Floor}}$$

However, since we are reading logarithmic units (dBm) from our spectrum analyzer the formula becomes:

$$\text{SNR} = \text{Signal Strength} - \text{Noise Floor}$$

To calculate between logarithmic and linear form we use the formula:

$$\frac{S}{N} = 10^{\frac{\text{SNR}}{10}}$$

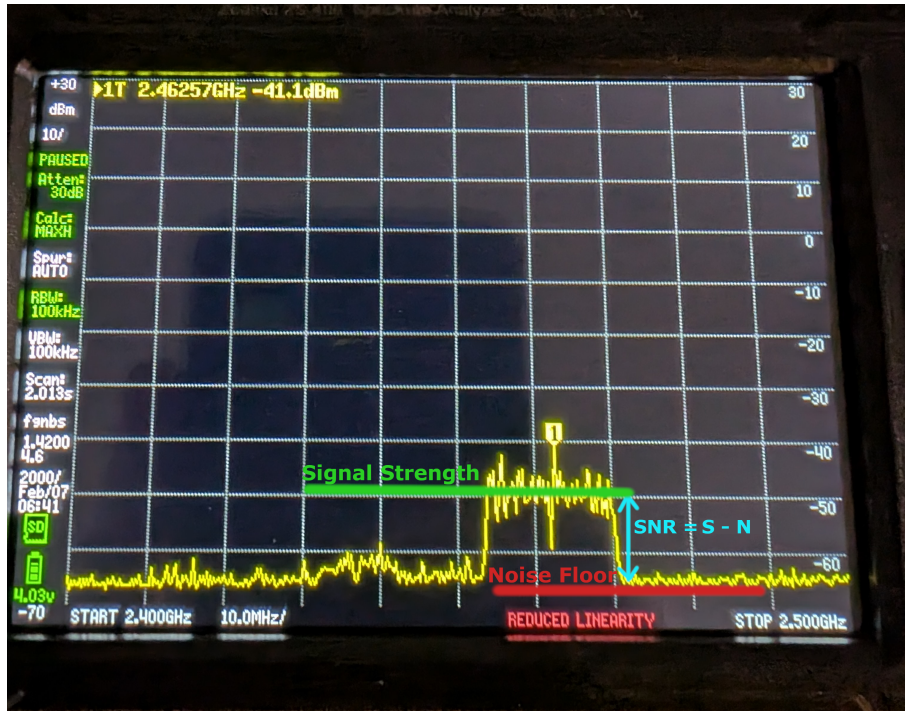


Figure 2.4: Spectrum analyzer displaying signal strength (green), noise floor (red), and calculated Signal-to-Noise Ratio (cyan) for a 2.4 GHz signal.

To keep our signal from being corrupted, we want our signal strength to be as far above the noise floor as possible. However, maximizing the signal strength has several drawbacks, such as more power usage, more internal noise from excessive amplification, interference towards other equipment and potential violation of regulatory limits. Therefore, a minimum necessary transmitter strength is chosen so that our signal can reliably reach the receiver.

2.2.2 WiFi Adapter

Using a WiFi enabled computer running a Linux distribution, we can find nearby wireless networks. To make themselves available for incoming connections, wireless access points broadcast management frames. These are used for clients to find, authenticate, and associate with the access points. By capturing these management frames we can get an indication of signal strength and in some cases the noise floor for the WiFi channels.

Although this signal strength is only an indication, we can compare available access points to our spectrum analyzer measurements to get a clearer picture of the RF environment.

We can use the Linux utility `iw` to capture management frames and probe responses. The command returns BSSID, SSID, channel, signal strength, capabilities, and more for all access points in range.

```
iw dev <interface> scan
```

2.2.3 Antennas

Free-space path loss [2.3.4](#) is most prominent in omnidirectional antennas, where the signal is spread evenly around the horizontal plane of the antenna. The characteristics of an antennas shape and direction of signal propagation is called a radiation pattern.

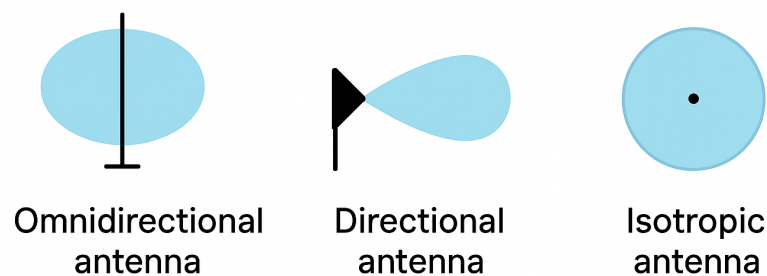


Figure 2.5: A single-plane simplification of radiation patterns for antennas.

Another important characteristic of an antenna is the antenna gain, often given in decibels relative to isotropic (dBi). A directional antenna typically has a much higher dBi value than its omnidirectional counterpart because it focuses its energy

into a narrow beam rather than spreading it out, as can be seen in Figure 2.5. An isotropic antenna is a theoretical antenna that radiates uniformly in all directions and has a gain of 0 dB. Real world antennas use the isotropic antenna as a reference point to indicate how much of the antenna’s signal is concentrated in a particular direction. The dBi value of an antenna is given for the direction of maximum radiation (or the most “forward” direction) of the antenna.

2.3 Effects on Wireless Channels

2.3.1 Noise

Noise ($\mathcal{N}$) is the combination of all unwanted signals that interfere with the signal ($\mathcal{S}$) we are trying to process at some receiver. This is a fundamental issue in all systems and cannot be completely eliminated. These source range from internal noise from the random, rapid motion of electrons within any resistive component (electronic components), to the external noise from cosmic, atmospheric and man-made noise, from which the latter is mainly made by other communication systems.

The primary measure of a wireless channel quality is the ratio of noise to the signal strength. This is often given in signal-to-noise ratio ($\frac{\mathcal{S}}{\mathcal{N}}$), or Signal-to-Interference-plus-Noise Ratio on more crowded bandwidths.

2.3.2 Timing

There are two main concerns for timing in a wireless communication system, the carrier frequency offset and the sample timing offset.

- Carrier frequency offset occurs because the oscillators for the source and receiver are never perfectly identical, leading to constellation point differences.
- Sample timing offsets make the receiver take samples at non-ideal points in time, where interference from close symbols can happen. For both of these

If using coherent modulation, both of these effects will drift out of control over time and must be estimated and corrected by the receiver to ensure a usable channel link.

2.3.3 Fading

Fading is the effect of signals taking multiple paths to the receiver and in effect components of the same signal arrive at different times, and at different signal strengths. The physical phenomenon of signals taking several paths to a receiver is often referred to as multi-path propagation, while fading is the resulting distortion on the received signal.

There are generally two types of stochastic models for fading, Rician fading and Rayleigh fading.

- Rician fading is used when there is a dominant direct line of sight path for the signal, with some scattered components. This type of fading is often seen in inter-satellite links or in between radio towers.

- Rayleigh fading is used when there is no dominant line of sight and signals received are the product of many reflected and scattered signals. This is typically seen in communication links in heavy built-up urban areas or indoors.

2.3.4 Path Loss

Like ripples in a pond, a signal travels in a straight line, but its energy spreads out across an ever-increasing area. This causes the signal strength to decrease with distance from the transmitter. This reduction in signal strength alone is known as free-space path loss. Another source of path loss is the reflection, scattering, and absorption of a signal as it travels between the source and destination.

For a distance d , a path loss exponent α and a constant representing a reference distance measurement B , free-path loss follows the power law relationship:

$$B * d^{-\alpha}$$

With PL = Path loss, d_0 as a reference distance, and α as a path loss exponent dependent on the environment, the power law relationship is transformed into a linear equation called the log-distance path loss model:

$$PL_{dB}(d) = PL_{dB}(d_0) + 10 * \alpha * \log_{10}\left(\frac{d}{d_0}\right)$$

Shadowing

Shadowing is the path loss effect of multi-path propagation happening on the large scale, and occurs when large objects like mountains and buildings create multiple paths between receiver and transmitter. While the fading following the Rayleigh or Rician models are often referred to as small-scale fading, shadowing is called slow fading. When doing signal strength measurements this effect will occur as discrepancies to the power law relationship and follows a log-normal difference in path loss, resulting in the path loss model:

$$PL_{dB}(d) = PL_{dB}(d_0) + 10 * \alpha * \log_{10}\left(\frac{d}{d_0}\right) + X_{\sigma}$$

Where X_{σ} is a log-normal variation determined experimentally for a given environment.

2.4 Simulation and Implementation Tools

2.4.1 Software-Defined Radio

A software-defined radio is a communication system where many of the components that are traditionally implemented as analog hardware are replaced by software controlled components. This allows for processing signals using software, replacing things like analog filters, mixers, modulators and demodulators.

2.4.2 GNU Radio

GNU Radio is a free, open-source software development kit designed to control software-defined radios, but can also be used to simulate signal processing and wireless communication. It provides a comprehensive library of existing signal processing blocks and allows for high flexibility by allowing custom coded blocks. GNU Radio documentation and manual can be found in the GNU Radio Wiki [9].

2.4.3 GNU Radio Companion

GNU Radio has a graphical user interface called GNU Radio Companion (GRC). A GNU Radio project can be visualized in GRC with a flowgraph, giving a user friendly way to quickly connect together signal processing blocks to define a data flow. Software is written in C++ for performance critical tasks while tools and flowgraph creation can be done in Python.

Flowgraphs

In the flowgraph, we simply place our blocks and connect them according to the network in mind. We simply drag and drop between the input and output ports of two blocks to connect them. The data type of an input or output is constant and must match the two connecting ports. The data type is marked by a color on the port and is usually represented as real or complex floating point numbers.

Blocks

Blocks are typically written in C++ for a performance advantage over those written in Python. Many standard processing blocks are already included in GNU Radio and it is easy to quickly prototype and test new signal processing algorithms using these, often in combination with self-made custom blocks. Some core block types are:

- **Source blocks:** Usually connected to a hardware component (SDR) but can also be used to pull data from files or generated sources.
- **Sink blocks:** Usually connected to a hardware component (SDR) but can also be used to visualize input signals or do some end processing (send data to file or other software) of a received signal.
- **General purpose blocks/processor blocks:** These include filters, mathematical operations on signals, modulation, and demodulation.

Out-of-Tree Modules

An out-of-tree (OOT) module is a GNU Radio component that is not already provided in the default GNU Radio installation. This allows users to create custom coded blocks and is the reason for the combination of ease of use and high flexibility of GNU Radio. These blocks can be created in Python or C++, and are configured using a yaml file.

2.5 Related Work

The core of calculations in this research ultimately stems from Shannons seminal work on the capacity of information channels in [3]. What follows directly is Cover and El Gamals work from 1979 which introduces the cut-set upper bound of the cooperative single-relay network in [2], and is also where the decode-forward and compress-forward were first introduced. While newer schemes have appeared such as the quantize-map-forward scheme presented in [8], most research have been done on improving the already existing schemes (e.g. [6] [7]).

A very good source of information on relaying schemes and a collection of formulas for the different schemes are found in the book "Network Information Theory" [5], where El Gamal is one of the authors.

Although the GNU Radio Wiki [9] is a bit messy with their different versions for C++ documentation, the wiki is a great source for learning about signal processing and has many example flowgraphs for this. The wiki also contains a list of selected papers referencing GNU Radio, this can be found in [10].

While doing research on the topic of relaying schemes most researchers tend to use simulations for their capacity estimates (e.g. [6] [7]), and in the few cases where an implemented testbed solutions is used, this is often the core of their research ([11] [12]). There are a few other frameworks than GNU Radio for creating testbed experiments, but I found none that give explicit instructions on how to conduct such experiments.

Chapter 3

Methodology

This chapter contains an explanation of methods used to configure the specific tools and how to harvest real-world empirical data. Following is the logic and math behind creating the code for visualizing the effect of relay placement using channel capacity formulas and empirical data. Finally the chapter concludes with a guide on how to use the measured empirical data in GNU Radio simulations, and finishes with examples on how to implement an almost complete solution in GNU Radio using software-defined radios.

3.1 Data Collection

3.1.1 Spectrum Analyzer

The TinySA is a product line of handheld spectrum analyzers. It is one of the most affordable spectrum analyzers, and the cost of a TinySA ranges from \$50 to \$250, making it very available for students and hobbyists. For comparison, the cost of professional analyzers ranges from \$5,000 to over \$100,000 (mid-range professional to high-end/research grade).

TinySA Ultra ZS-406

I am using the TinySA Ultra ZS-406 (see Figure 2.4). This analyzer offers a frequency range of 100 kHz to 900 MHz in “normal mode”. The frequency range can be extended to 5.3 GHz in what they call “ultra mode”, which covers our use case of the 2.4 GHz ISM-band. It can be controlled as a standalone device using the touch screen and physical buttons, but it can also be connected to a computer using Serial over USB with a set of SCPI-style commands. It has a resolution bandwidth of 200 Hz to 850 kHz, an internal attenuator of 0 to 32 dB and a built-in low-noise amplifier with 20 dB gain (effective up to 4 GHz). Documentation can be found on the TinySA website [13].

Calibration

To make sure our device is not damaged and working as intended we start by calibrating our spectrum analyzer. We connect a SMA cable between the CAL and RF ports before starting the calibration. The self tests goal is to check the internal health and functionality of the spectrum analyzer, while the level calibration is done

to maximise the accuracy of the dBm power level readings.

Self test and level calibration can be completed by running:

Menu -> Config -> Self Test and

Menu -> Config -> Level Cal, respectively.

Enable Ultra Mode

To support the high frequencies of 2.4 GHz WiFi we need to first set our tinySA to ultra mode. To do this, navigate to:

Menu -> Config -> More -> Enable ultra mode

and insert the passcode (defaults to 4321). This will increase the frequency range of our device from 100 kHz-900 MHz to 900 MHz-5.4 GHz. Note that ultra mode also increases the input sensitivity, therefore you should be careful when using large antennas or when in the presence of strong signals.

When you put the tinySA in ultra mode you may see false signals from spurs or mirrors and it is recommended to connect an external filter to remove these. Although we do not have a dedicated hardware filter, our directional antenna works as both a spatial filter and a frequency filter. TinySA comes with a spur removal algorithm, but with a directional antenna or filter it might be beneficial to turn the spur removal off, as this will increase scan speed and improve detection of short-duration signals. Spur removal can be turned on/off in:

Menu -> Config -> Spur removal

Frequency Settings

Our antenna is specified for WiFi in the 2.4 GHz ISM-band and so we are limited to measuring SNR of WiFi access points.

All these settings are changed in the frequency menu:

Menu -> Frequency

For the whole 2.4 GHz bandwidth set the **start frequency** to 2.4 GHz and the **stop frequency** to 2.5 GHz. Setting both start and stop frequencies will automatically set our center and span frequencies.

For reading a single WiFi channel we can also set our start and stop frequencies like this. However a simpler approach may be changing the center frequency according to Table 3.1, and set our bandwidth span to either 20 MHz or 40 MHz, depending on the network.

Table 3.1: IEEE 802.11 b/g/n 2.4 GHz WiFi Channels and Center Frequencies

Channel Number	Center Frequency (MHz)
1	2412
2	2417
3	2422
4	2427
5	2432
6	2437
7	2442
8	2447
9	2452
10	2457
11	2462
12	2467
13	2472
14	2484

Source: Data derived from the IEEE 802.11 Standard for Wireless Local Area Networks. [14]

Resolution Bandwidth

Since noise power is proportional to the resolution bandwidth (RBW), we want to have a narrow RBW. I set my RBW value in the 10 kHz to 30 kHz range.

Menu -> Frequency -> RBW

This will give us a better representation of the noise floor, although it increases our sweep time significantly.

Video Bandwidth

We set our Video bandwidth (VBW) to one of the smallest values available, such as 0.03 or 0.01 RBW. This will significantly increase the sweep time, especially since we have also chosen a narrow RBW. This does not reduce the actual noise, but will smooth it out and reduce variation in the visible noise floor. This setting can be changed in:

Menu -> Frequency -> VBW

Attenuation

The tinySA has an internal attenuator at max 30 dB, but since we are measuring unknown signals we start a measurement by connecting an external attenuator of an additional 30 dB to our RF input. Then we reduce our attenuation until we get a signal that is close to the maximum of what the TinySA can handle, which is 10 dBm. If we need our external 30 dB attenuator while measuring, we set the offset (or external gain) to -30 dB.

Menu -> Level -> Ext gain

We can let the tinySA automatically choose internal attenuation or we can manually change it in:

Menu -> Level -> Attenuate

When only using the internal attenuator we do not need to change the ext gain setting.

Spectrum Analyzer Measurements

WiFi is not a continuously transmitted signal and instead uses a type of multiplexing which transmits in short bursts on a narrow part of its channels bandwidth. This can be seen in Figure 3.1, where bursts covering almost the whole channel bandwidth has been measured. To capture these short bursts on the spectrum analyzer we change our trace to hold at maximum measured peaks with:

Menu -> Trace -> Calc -> Max Hold

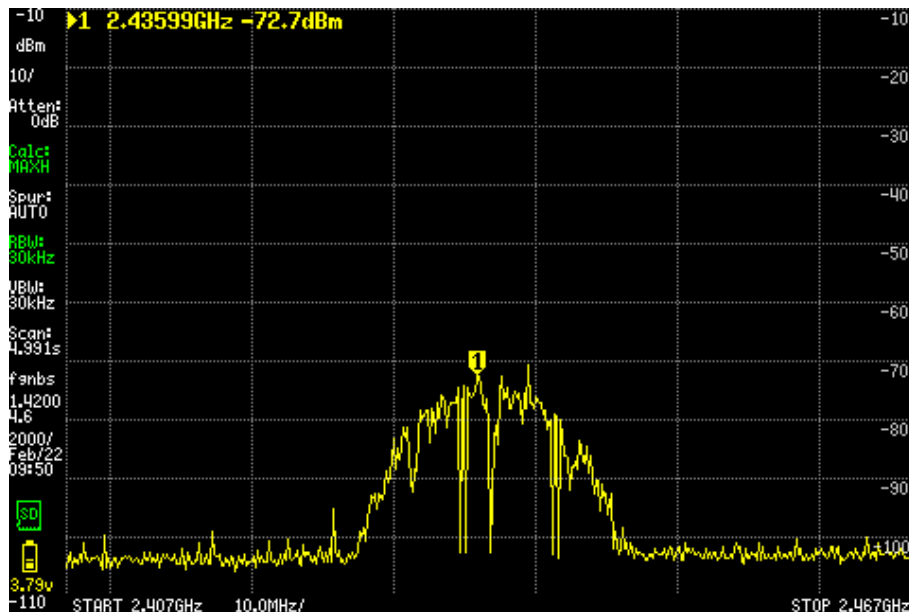


Figure 3.1: A measured WiFi signal where some parts of the total bandwidth have not been measured yet.

After logging the peaks for a while (depending on network activity), we simply move our marker to the peak of our wanted signal to get the signal level and then move it to the weakest signal strength part of the measured bandwidth to get a noise floor measurement.

3.1.2 WiFi Adapter

Most people use the built-in WiFi capabilities of their computer, but for a few people with ‘special interests’, it is not enough. I am using the Alfa Networks *AWUS036ACHM* (MediaTek MT7610U chipset) and like any other WiFi adapter it follows the 802.11 standard, more specifically the 802.11ac, with full backward compatibility. For those with special interests, it supports both monitor mode and packet injection. But for our use case, the most important feature is the RP-SMA

female port, allowing us to connect an external antenna to the adapter.

To make sure I am measuring the correct WiFi access point, I wrote a Python script (5.2) to give an overview of nearby wireless access points and their channels, giving an output similar to Figure 3.2.

```
hansi@Thinkpaden:~/Documents/Master-s-stuff/wifi$ sudo python3 wifiScript.py
[sudo] password for hansi:
Executing: iw dev

Available wireless interfaces:
  1) wlp61s0
Please select an interface (1-1): 1
Interface selected: wlp61s0
Executing: sudo iw dev wlp61s0 scan 2>/dev/null

--- Detected Wi-Fi Networks ---
SSID: Fana DPS
- Protocol: 802.11b/g
  RSSI: -64.0 dBm
  Channel: 11
  Frequency: 2462 MHz
  Security: WPA2
- Protocol: 802.11ac (Wi-Fi 5)
  RSSI: -69.0 dBm
  Channel: 52
  Frequency: 5260 MHz
  Security: WPA2
```

Figure 3.2: Output of my WiFi scan Python script.

We can also use the **airodump-ng** command as shown in 3.3. However, with monitor mode enabled, this will capture all raw 802.11 frames, which might raise some privacy concerns. Code to set up monitor mode can be found in 5.2.

```
CH 13 ][ Elapsed: 1 min ][ 2025-10-17 17:32
```

BSSID	PWR	Beacons	#Data, #/s	CH	MB	ENC CIPHER	AUTH	ESSID
D8:EC:E5:E8:B3:65	-1	0	0 0	12	-1			<length: 0>
36:12:9E:53:A1:04	-82	2	0 0	11	648	WPA2 CCMP	PSK	<length: 0>
36:12:9E:23:A1:04	-82	2	0 0	11	648	WPA2 CCMP	PSK	My home
88:41:FC:F3:31:D5	-80	0	2 0	11	-1	WPA		<length: 0>
34:21:09:94:13:D1	-82	2	0 0	6	130	WPA2 CCMP	PSK	Omni V2_9413C0
D8:EC:E5:F0:7A:05	-81	2	0 0	11	260	WPA3 CCMP	SAE	Zyxe1_7A05

Figure 3.3: Output of airodump-ng (requires monitor mode).

3.1.3 Antenna

The antenna I use can be seen in Figure 3.4, and is a parabolic antenna ordered from China. It is branded as ‘SignalPlus SD-2425-15D’, and is in the sub \$50 price range. It has a gain of 15 dBi, a frequency range of 2400-2483 MHz, and a nominal impedance of 50 Ω , matching both my spectrum analyzer and WiFi adapters input impedance.



Figure 3.4: A Chinese parabolic WiFi antenna.

Other Components

The parabolic antenna uses an SMA connector and matches the spectrum analyzers connector. The standard for consumer WiFi devices is RP-SMA and I needed an adapter to connect to the WiFi adapter. The only difference between SMA and RP-SMA connectors is that the center pins are reversed between the male and female connectors. I also have a “RF Coaxial Attenuator SMA Male to Female DC-6 GHz 2W 50 Ω fixed attenuator”, *also from China*, to protect my spectrum analyzer when measuring strong or unknown signals.

- Parabolic antenna: SMA male (center pin)
- Spectrum Analyzer: SMA female (center hole)
- WiFi adapter: RP-SMA female (center pin)

3.1.4 Real-World Measurements

For these measurements I used a TL-MR6400 TP-link router in Figure 3.5. The choice of WiFi channel is usually set automatically by routers, but after checking for the least occupied channels with my spectrum analyzer I changed the router to use channel 6. To increase the transmit distance I set the bandwidth down to 20 MHz. I also set the transmit power of the router to “high”, whatever that means. I confirmed with the Python WiFi script (5.2) that I was not measuring some other access point.



Figure 3.5: TP-link router in window.

The first measurement was just outside the window, approximately 7 meters from the router. I set the bandwidth span to 60 MHz and the center frequency to match WiFi channel 6 in Table 4.1. This should give us a nice view of our 20 MHz signal in the middle and we will be able to see the noise floor on one of the bordering 20 MHz.

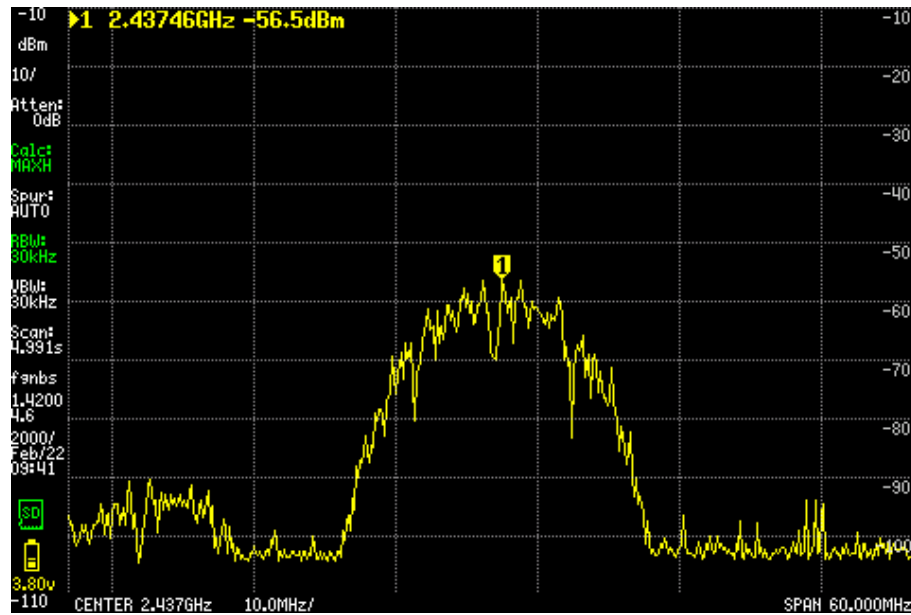


Figure 3.6: TinySA screen showing a 20 MHz signal measured from the location of Figure 3.7



Figure 3.7: Measurement point: 7 meters from the router.

I move further from the router and take another measurement, this time 50 meters away. For each location I save a bitmap of the tinySA view with:

Menu -> Storage -> Save capture



Figure 3.8: Measurement point: 50 meters from the router

I continue like this for three more measurements and save the bitmaps on my tinySA spectrum analyzer, and note down the location using the GPS on my phone. These distance measurements do not take into account elevation, but at these small elevation differences the difference in distance would most likely be smaller than GPS-location inaccuracies of my phone.

I also pointed the antenna away from any housing and got a noise floor reading of -110 dBm. To get an overview of the measurements I created a map using the image processor GIMP and marked each location. Using the formulas from chapter 2, I created a table of distance, signal strength, SNR, S/N and theoretical Shannon capacity for each measurement.

3.1.5 Testbed Data

In [12] they use a WARPLab [15] testbed to implement amplify-forward, decode-forward, and quantize-map-forward relaying schemes. An LDPC code of rate $\frac{3}{4}$ with codeword length of 1944 bits and 16 Quadrature Amplitude Modulation(QAM) is used. This is an indoor testbed and has three different node placement scenarios. The results they present are RSSI for each link, but also the frame error rates and throughput in bits/Hz for each of the three scenarios. The scenarios are briefly explained as:

Scenario 1: The Source-Relay link is much stronger than the source-destination and relay-destination.

Scenario 2: All three links have approximately equal signal strength.

Scenario 3: The source-relay and relay-destination links are much stronger than the source-destination link.

Table 3.2: Testbed performance Comparison of Cooperative Relaying Schemes

Scenario 1		Scenario 2		Scenario 3	
Link	dBm	Link	dBm	Link	dBm
S-D	-76.2	S-D	-78.1	S-D	-69.7
R-D	-77.0	R-D	-79.3	R-D	-68.8
S-R	-59.0	S-R	-76.2	S-R	-67.4
Scheme	bps/Hz	Link	bps/Hz	Link	bps/Hz
AF	1.12	AF	0.84	AF	0.85
DF	1.20	DF	0.95	DF	0.88
QMF	1.22	QMF	1.06	QMF	0.89
Direct	1.00	Direct	0.76	Direct	0.06

Table 3.3: Simulated performance of QMF using 16 QAM modulation

SNR_{SD}	SNR_{RD}	SNR_{SR}	bps/Hz
0	0	10	1
4	4	14	2
8	8	18	3.2
12	12	22	3.8
16	16	26	4

3.1.6 Simulated Data

Simulated data for QMF using 16, 64 and 256 QAM is presented in [16]. I will extract the data using 16 QAM modulation to match the data from the testbed results in [12]. However, the paper finds that QMF is most efficient when using 64 QAM and 256 QAM, and has its information theoretic thresholds at 14.18 dB and 13.47 dB respectively. A continuous graph is presented for the results, in which i take some samples and add to Table 3.3. As the SNR_{SD} and SNR_{RD} reach about 10 dB the gain from increasing the dB further starts to fall off.

3.1.7 Normalizing the data

While the simulated data given in signal-to-noise ratio, the testbed data is presented in signal strength (dBm). To compare the data, we need to make some assumptions about the noise floor level. From our real world measurements we found a noise floor of about -100 dBm in a rural outside location, and I must assume a testbed lab has a rather low noise floor reading. However, I will use a range of noise floor from -80 dBm to -100 dBm. I will use the data from scenario 1 in the experimental study, since that is what most closely resembles the SNR link differences in the simulated study. The data is calculated and added to Table 3.4.

Table 3.4: Experimental performance of QMF in scenario 1 with different noise floors

Noise floor	SNR _{SD}	SNR _{RD}	SNR _{SR}	bps/Hz
-80	3.8	3	21	1.22
-90	13.8	13	31	1.22
-100	23.8	23	41	1.22

3.2 Placement Based Relay Channel Capacity

Using the real-world measurement from 3.1.4 I create a formula for SNR given distance, similar to path loss given distance. To do this I fit the data to the power law formula (see 2.3.4) using non-linear least squares fitting. After calculating the values from Table 4.1 in Python, I use the *scipy.optimize module* within the 5.2 Python script to do non-linear least squares fitting of data to a function. I later do a similar, but more hands-on calculation using only a single reference measurement in section 3.3.2.

Assume now that my source and destination node are in a (x, y) grid where each square in the grid is of size $1m^2$. I place our destination and source in the middle of the y-axis, then from the middle of the x-axis we move each point away by $distance/2$ in opposite directions. I place a relay in any position on the grid and calculate the euclidean distance to this point, approximate a signal-to-noise ratio using the fitted power rule, and finally calculate the cooperative capacity using the cut-set upper bound from 2.1.4. Repeating this for every position in the grid, I use the Python library *matplotlib* to display a heatmap of relay capacities.

After creating a heatmap using one of the measured values from section 3.1.4, I use GIMP to overlay it on a real-world map and align it with the source and destination locations marked in Figure 4.3.

Finally I add the relay-specific capacity formulas for amplify-forward, decode-forward, and compress-forward (see 2.1) to the Python code to be able to create heatmaps for each relaying scheme.

3.3 Simulation and Implementation in GNU Radio

This section will work as a guide on how to create a simulation or a testbed implementation of a wireless communication channel or network in GNU Radio Companion (GRC). I will show how to implement out-of-tree modules which allows for the creation of custom blocks, and how to simulate naturally occurring effects of wireless communication using existing GNU Radio blocks. Finally, I take a look at converting the simulation to an SDR implementation.

3.3.1 Custom Blocks

To simulate the work of the source, relay and destination I create three new blocks. To do this I first create a GNU Radio out-of tree (OOT) module, and choose C++ as the language for its performance advantage over Python. The following commands generates the file structure and boilerplate code needed for development. After installing the necessary libraries for creating OOT modules I run:

```
gr\_modtool newmod <moduleName>
```

To generate a new custom block within the module, run:

```
gr\_modtool add <blockName>
```

In the new modules folder we find the *lib* folder which contains our boilerplate code for the custom blocks.

The code of *<blockName>.cc* has a private constructor where we set the output and input ports of the block, and for the source block it looks like this:

```
1 SourceSimulated_impl::SourceSimulated_impl()
2   : gr::sync_block("SourceSimulated",
3                     gr::io_signature::make(0, 0, 0),
4                     gr::io_signature::make(1, 1, sizeof(
5                       output_type)))
6 {
7   message_port_register_out(pmt::mp("out"));
8 }
```

I have assigned *output_type* as *std::complex<float>*, which is the standard for representation of physical signals in GNU Radio.

The numeric arguments of *gr::io_signature::make(1, 1, sizeof(output_type))* correspond to the minimum amount of output ports, the maximum amount of output ports, and the size per item in bytes, respectively. The prior line is the same but for input ports. I add a message port to send bits to some destination block for comparing received bits with sent bits by adding the line:

```
message_port_register_out(pmt::mp("out"));
```

Finally we have the work code block. This function is called continuously by the GNU Radio scheduler and does all our work of handling inputs and outputs.

In the example code below some bits are generated and added directly to the output port. This code is not compatible with our previous complex output configuration.

```
1 int SourceSimulated_impl::work(int noutput_items,
2                                gr_vector_const_void_star&
3                                input_items,
4                                gr_vector_void_star& output_items
5                                )
6 {
7   auto out = static_cast<output_type*>(output_items[0]);
8
9   // Use a uniform integer distribution for random numbers
10  between 0 and 1
```

```

8     std::uniform_int_distribution<int> distribution(0, 1);
9
10    // Generate random bits and fill the output buffer
11    for (int i = 0; i < noutput_items; i++) {
12        out[i] = distribution(d_rng);
13    }
14
15    // Example debug output
16    GR_LOG_INFO(d_logger, "Exiting work() function.");
17
18    // Tell runtime system how many output items we produced.
19    return noutput_items;
20 }

```

The `/grc/*.yml` file determines GUI configurable parameters and how the block shows up in gnu radio (e.g. port names)

To build the module I run `cmake .` then `make` to compile and finally `sudo make install` to make the blocks available in GNU Radio companion.

To rebuild after making changes I simply run `make` and `sudo make install`.

3.3.2 Flowgraphs and In-Tree Module Blocks

To model any type of network structure in GNU Radio we use the GNU Radio companion (GRC). Using the custom source, relay and destination blocks, the most simple form of simulation in GRC can look like Figure 3.9.

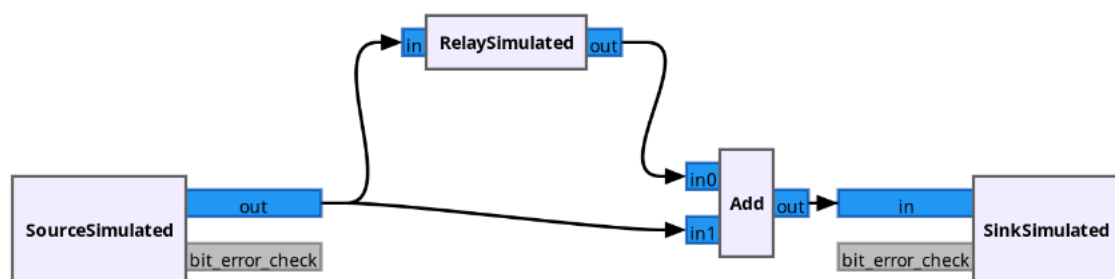


Figure 3.9: A simplistic GNU Radio simulation visualized in GNU Radio companion.

In addition to the node blocks, an add block is used to superimpose the signals from source and relay to the input of the destination node. Several of the following wireless effects can be introduced using the in-tree channel model block, while some need to be implemented more explicitly.

Noise

The simulation in Figure 3.9 does not inherently include noise. To generate noise I use the *channel model block*, which in addition to other things generates gaussian white noise and adds it to the passing signal. Figure 3.10 shows the channel model block being used over a single wireless channel. The signal will now be changed according to the effects configured when passing through the channel model block. If left un-configured the channel model block will do nothing to the signal.

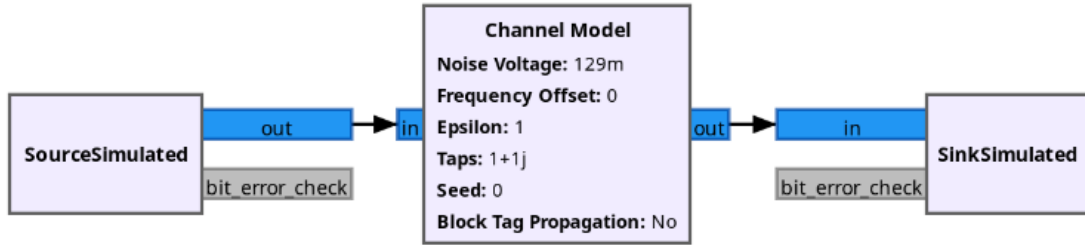


Figure 3.10: A single wireless channel with noise in GNU Radio companion.

I configure the noise of this block by setting the "Noise Voltage" parameter. This value sets the standard deviation for both the real and imaginary part of the complex signal passing through. Assume the target linear signal-to-noise ratio is $S/N = 30$ and the average power of the input signal is $P = 1$ (standard for simulations). Our Noise voltage setting should then be set to $\sqrt{\frac{P}{2*S/N}} = 0.129$.

Timing

The channel model block can do both frequency and sample timing offset.

- With the frequency offset parameter we can set the difference between the transmitter and receivers' carrier frequencies. The value is normalized and a value of 0.25 would be one quarter of the symbol rate, while a value of 0 is no difference. For a frequency offset parameter ν and a symbol rate R_s . The carrier frequency difference Δf between transmitter and receiver is:

$$\Delta f = \nu * R_s$$

- The sample timing offset is the difference between the sample clocks of the transmitter and receiver and is set by the epsilon parameter. Assume a transmitter sampling frequency SF_{Tx} and receiver sampling frequency SF_{Rx} . The sample timing offset ϵ is given by:

$$\epsilon = \frac{SF_{Tx}}{SF_{Rx}}$$

Both of these values can be left as default to make the synchronization at the demodulation stage simpler. We generally do frequency synchronization first, then follow up with sample time synchronizations. Many blocks for synchronization are in the core library of GNU Radio and can be found under the "Synchronizers" category.

Fading

For fading I configure the already placed channel model block. This can be set using a static power delay profile (PDP), which is a list or vector of complex numbers. The first element of this vector is the line-of-sight path and is typically set to $1.0 + 1j$, each of the following elements of the vector is delayed according to its position in the vector (by one sample).

As an example we set the vector to be $[1 + 1j, 0, 0, 0.7 + 0.3j]$. The line-of-sight signal $(1, 1j)$ is untouched, while after three samples another signal arrives $(0.7 + 0.3j)$, which includes a phase shift in addition to a magnitude change.

We can use the Rayleigh or Rician models from 2.3.3 to set the values of the channel model PDP to create static fading, or the dedicated fading model block for dynamic fading.

Path Loss

There is no specific built-in block for path loss in GNU Radio, I instead use the existing general purpose block, the multiply const block. Path loss is a linear (non-dB) amplitude scaling factor, and we can simulate this by running our signal through a multiply const block and configuring it with a value calculated from some path loss formula, for instance the the power law formula from 2.3.4. With PL = Path loss, d_0 as a reference distance, and α as a path loss exponent dependent on the environment, the complete formula looks like this:

$$PL_{dB}(d) = PL_{dB}(d_0) + 10 * \alpha * \log_{10}\left(\frac{d}{d_0}\right)$$

For α I use the previously calculated loss exponent found in section 3.2, and set $\alpha = 1.87536666$. For the reference signal loss I use the signal strength difference between two of my measured values from section 3.1.4, more specifically at 7 meters and 50 meters.

$$PL_{dB}(d_0) = PL_{dB}(7) - PL_{dB}(50) = 16dB$$

This represents a 16 dB path loss after the signal has traveled 43 meters. Setting $d_0 = 50 - 7 = 43$ and substituting in my values gives a formula to calculate the decibel path loss given a distance:

$$PL_{dB}(d) = 16 + 10 * 1.87536666 * \log_{10}\left(\frac{d}{43}\right)$$

Since the multiply constant block is a linear gain, we need to convert the calculated decibel path loss (PL_{dB}) into a linear amplitude scaling factor (A). The linear power gain (G) is:

$$G = 10^{-(PL_{dB}/10)}$$

And the linear amplitude gain (A) is:

$$A = \sqrt{G}$$

We finally arrive at the formula:

$$\begin{aligned} A(d) &= 10^{\frac{-(PL_{dB}(d))}{20}} \\ &= 10^{\frac{-(16+10*1.87536666*\log_{10}(\frac{d}{43}))}{20}} \end{aligned}$$

The parameter field of the multiply const block does its calculation using Python, and we can add Python's math library by adding an import block to our flowchart and adding the line `import math` to the blocks configuration. The distance variable in the formula can be changed either manually or by using any of the variable blocks in GNU Radio. After converting the formula to something Python understands and renaming the distance variable to something useful, we end up with something I can put in the multiply const block directly:

$$\text{pow}(10, -(16 + 10 * 1, 8763666 * \text{math.log10}(\text{distanceSD}/43))/20)$$

This block is then placed between our transmitting block and the channel model block as shown in Figure 3.11.

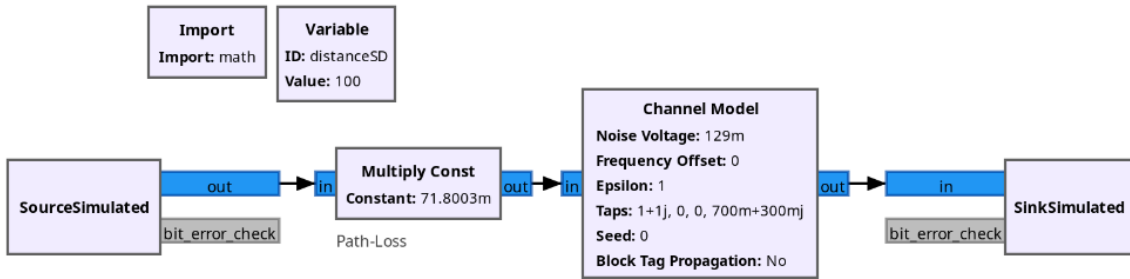


Figure 3.11: A single wireless channel with noise, fading and path loss in GNU Radio companion.

3.3.3 Single-Relay Channel Flowgraph

In Figure 3.12 I have created a flowgraph of the wireless effects for a single-relay channel. However, for this flowgraph to work, the simulated blocks (source, relay, sink) has to do all the required signal processing in the work function of their code.

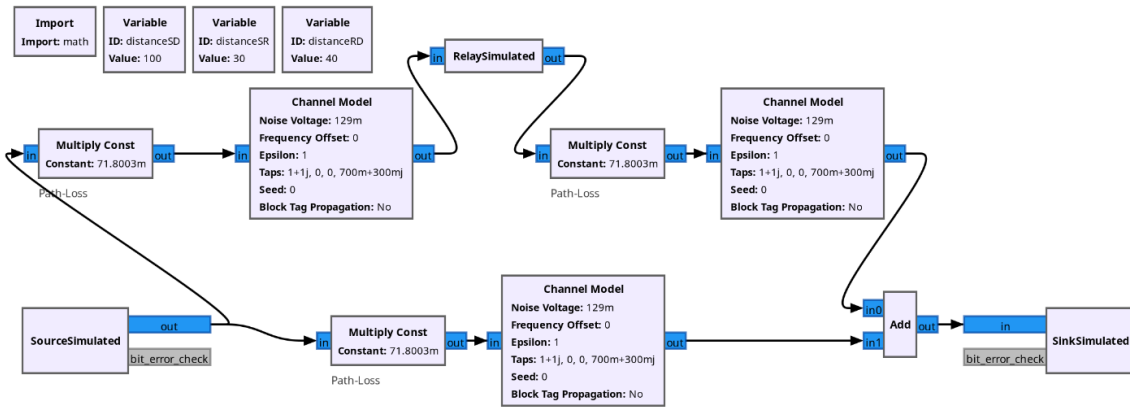


Figure 3.12: Single-relay channel with noise, fading and path loss in GNU Radio companion.

3.3.4 Software-Defined Radio Implementation

Assume we have a setup of three software-defined radios (SDR) to create the single-relay model. Each SDR is of the type USRP which are integrated into GNU Radio as source and sink blocks. Following the earlier assumption that the custom blocks

for source, relay and sink do all signal processing for us, we can easily convert the simulated example in Figure 3.12 to a working implementation. The figures in 3.13 show the simplicity of the three SDR implementation flowgraphs for source, destination and relay.

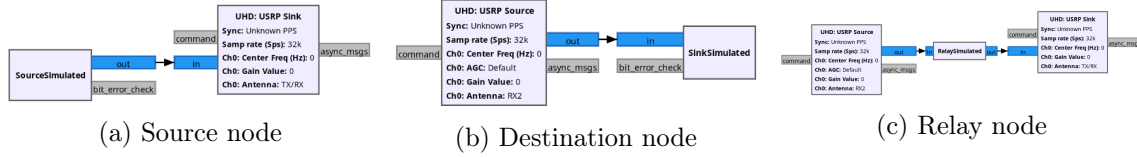


Figure 3.13: Flowgraph of SDR implementation for source, destination and relay nodes.

GNU Radio signal processing

While the custom blocks currently do all signal processing, we probably want to offload some of this to GNU Radios in-tree blocks instead. By changing the output and input types of all custom blocks to unsigned char or uint8_t (bytes). The custom blocks now only work as data sources and encoders/decoders. Data sources in GNU Radio can be done using pipes, read from files or generated randomly. Some coding blocks already exist as in-tree GNU Radio blocks, but will be lacking for a complex cooperative relay channel.

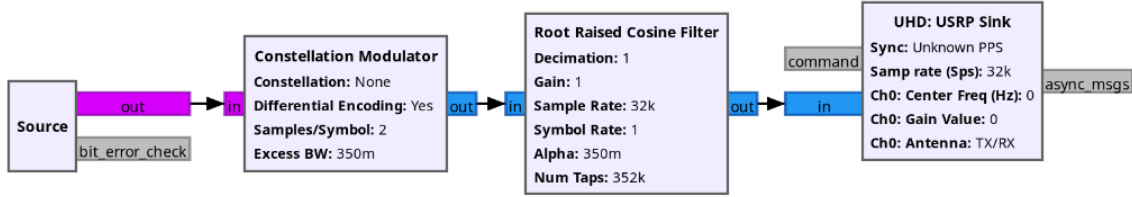


Figure 3.14: Flowgraph of SDR implementation for a source node with GNU Radio modulation and pulse shaping.

The constellation modulator takes in bytes and converts them to a complex signal stream, while the root raised cosine filter block is one of the ways of doing pulse shaping.

Chapter 4

Results

This chapter presents the data findings from Section 3.1.4, examples of the visualizations made by the code created in 3.2, and shows the resulting GNU Radio flowgraphs for simulation and implementation made in 3.3.

4.1 Data Findings

Section 3.1.4 can be used as a guide on how to configure a spectrum analyzer to make noise floor and signal strength measurements, as well as how to use a WiFi adapter to confirm the signal source. I also try to compare data found in simulated and experimental papers from Section 3.1.7.

4.1.1 Measured data

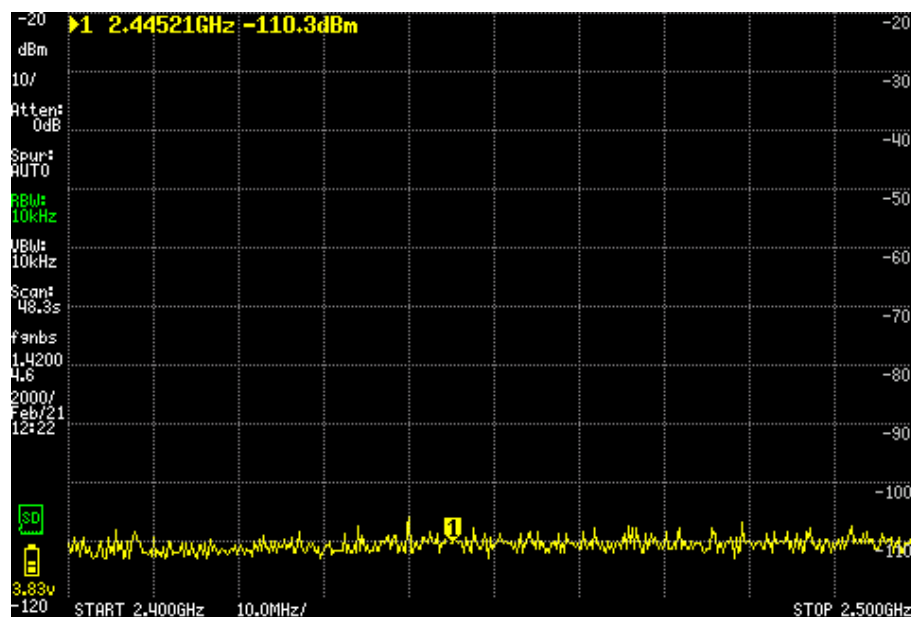


Figure 4.1: A noise floor reading far away from any noise sources.

I first took a measurement of the 2.4 GHz spectrum in a very secluded area (Figure 4.1). This measurement read a noise floor of -110 dBm, but as we can see from

the five readings in Figures 4.2, a more reasonable noise floor is somewhere between -110 dBm and -100 dBm.

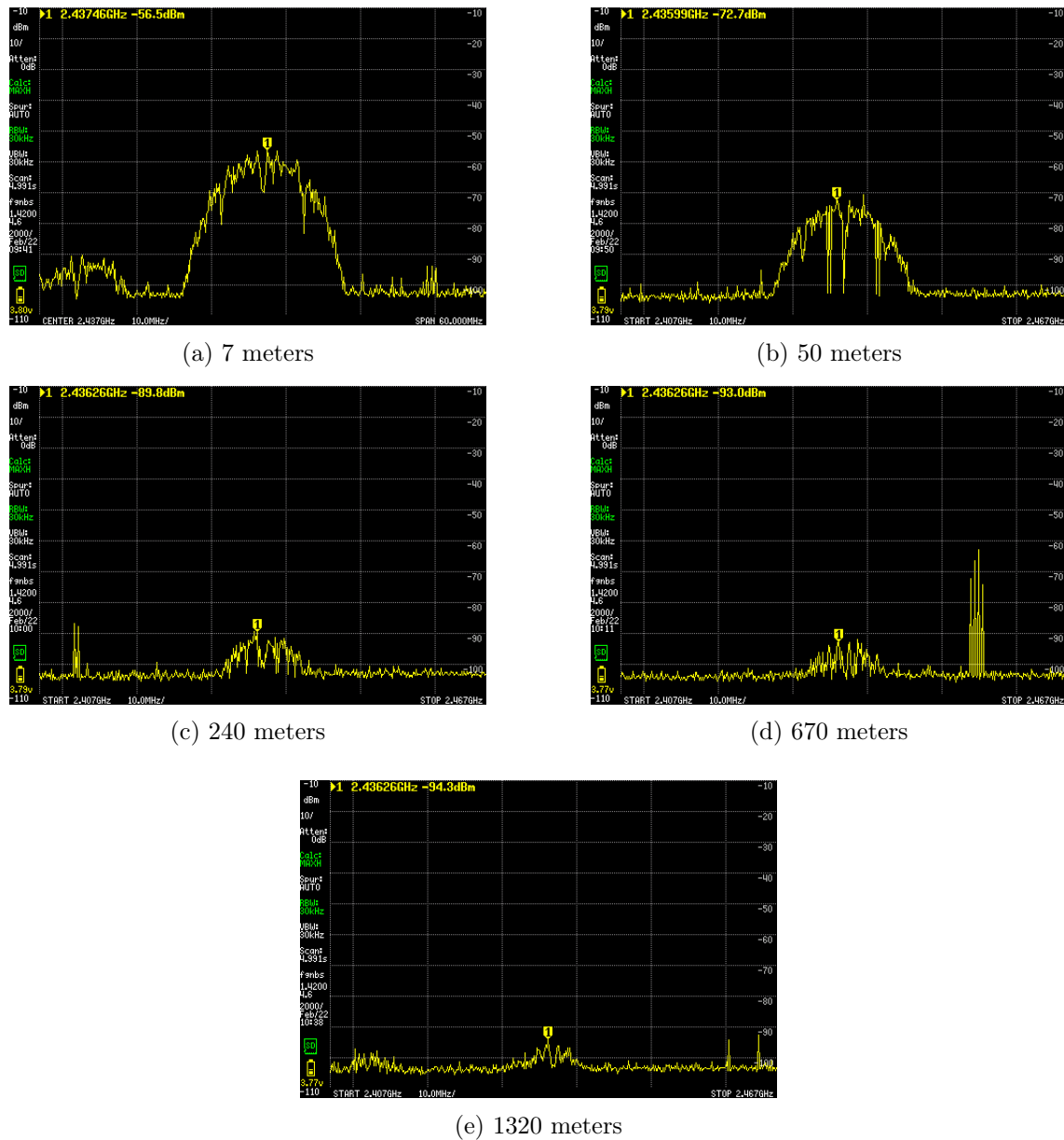


Figure 4.2: Screenshots from TinySA measuring our routers signal from 5 distances.

Reading off the signal strength values from Figures 4.2 and using a noise floor of -105 dBm I put the data and some calculations into Table 4.1. To get an overview of the measurements I created Figure 4.3 where I have marked each measuring location on a map and the corresponding SNR value.

Table 4.1: Measured Wireless Signal Parameters and Calculated Theoretical Capacities.

Distance (meters)	Signal strength S (dBm)	SNR (dB)	S/N (ratio)	Theoretical Shannon capacity $\frac{C}{B}$ (bps/Hz)
7	-57	48	63095.734	15.945
50	-73	32	1584.893	10.631
240	-90	15	31.623	5.028
670	-93	12	15.849	4.075
1320	-96	9	7.943	3.161



Figure 4.3: Map of SNR (dB) for each measurement location. Crossed circle (red) is the router, delta (teal) is a measuring location. Google, n.d. [1]

While fitting the data to a path loss formula later in the results section I find that the simple measurements of signal strength and noise floor do not capture the whole picture.

4.1.2 Simulated and Testbed Data

From the simulation data of Section 3.1.6, the SNR of the S-D and S-R links are equal, and because “The capacity advantage of cooperative relaying is most pronounced when the source to relay link is significantly better than the direct link between source and destination.” [16], the SNR of the R-D link is conveniently ten times stronger (10 dB) than the other links.

While the simulated data is given as throughput for a given SNR (dB), the experimental data from Section 3.1.5 uses some dubious techniques. First they only compare the different throughputs using signal strength, while no noise floor is given. In addition the signal strength is being measured by RSSI. RSSI is a received signal strength indicator and is not a precise measure of signal strength. I tried to compare

Table 4.2: Simulated performance of QMF using 16 QAM modulation

SNR_{SD}	SNR_{RD}	SNR_{SR}	bps/Hz
4	4	14	2
8	8	18	3.2
12	12	22	3.8
16	16	26	4

Table 4.3: Experimental performance of QMF given different noise floors

Noise floor	SNR_{SD}	SNR_{RD}	SNR_{SR}	bps/Hz
-80	3.8	3	21	1.22
-90	13.8	13	31	1.22
-100	23.8	23	41	1.22

the data to the best of my ability and picked a rather high noise floor for a testbed lab, more specifically -100 dBm to -80 dBm. Table 4.2 contains the simulated results and Table 4.3 contains the experimental results using three estimated noise floors.

Table 4.4 compares the simulated throughput with the theoretical formulas for the cut-set bound and for specific relaying schemes. keep in mind that the SNR values are chosen to maximize the effect of quantize-map-forward and should in theory be more related to the compress-forward (CF) throughput.

4.2 Relay Channel Capacity Visualization

The result of section 3.2 is a Python script used to visualize capacity based on relay placement. This script uses the empirical data found in section 3.1.4 for SNR calculations, and creates a 2D heatmap of different relay placements in the area between source and destination nodes.

After fitting the empirical data with the Python code in 5.2, I end up with this fitted power law formula:

$$S/N = 2,425,878.53 * d^{-1.87536666}$$

Table 4.4: Throughput (bps/Hz) for simulated quantize-map-forward and theoretical calculations for different schemes

SNR_{SD}	SNR_{RD}	SNR_{SR}	simulated	cut-set	AF	DF	CF
4	4	14	2	3.7	3	3.2	2.9
8	8	18	3.2	4.5	3.8	4.1	3.7
12	12	22	3.8	5	4.4	4.5	4.3
16	16	26	4	5.3	4.7	4.8	4.6

Using this formula and calculating the values back to SNR (dB) gives us the fitted model shown in Figure 4.4. For a perfect single free-path channel the measured (red) points should in theory fit the path loss model perfectly, and the discrepancy in the model is likely an effect of shadowing (see 2.3.4). As this is measured in a very rural area with mostly free line-of-sight path loss, this strongly indicates that the measurement of only signal power and noise floor is lacking for real-world scenarios. Keep in mind that multi-path propagation will be much more prominent in urban areas.

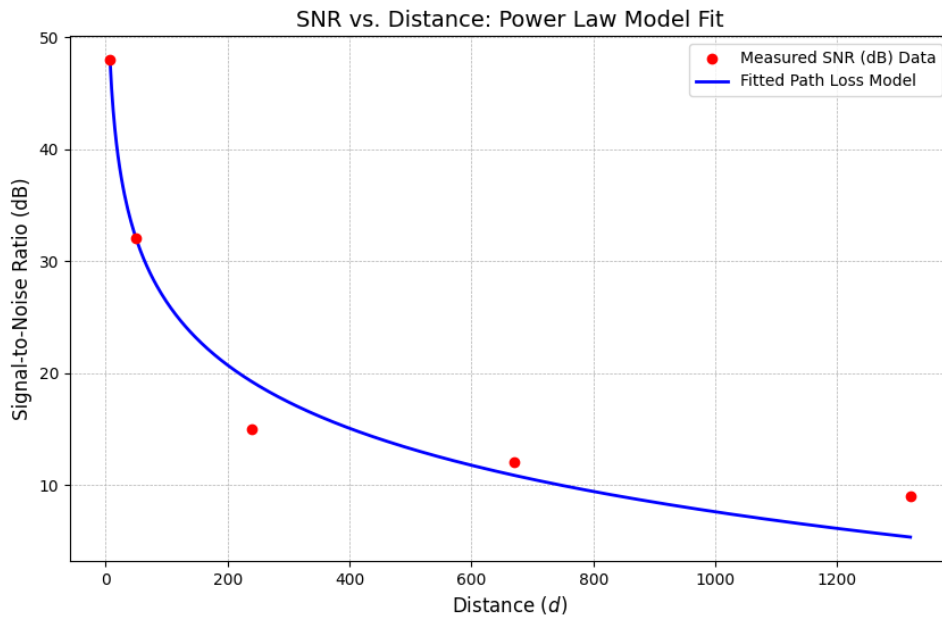


Figure 4.4: Fitted path loss model vs measured data points in SNR.

Figure 4.5 shows a capacity heatmap of the cutset-upper bound for a 1320 meter source-to-destination distance, using the code created in section 3.2.

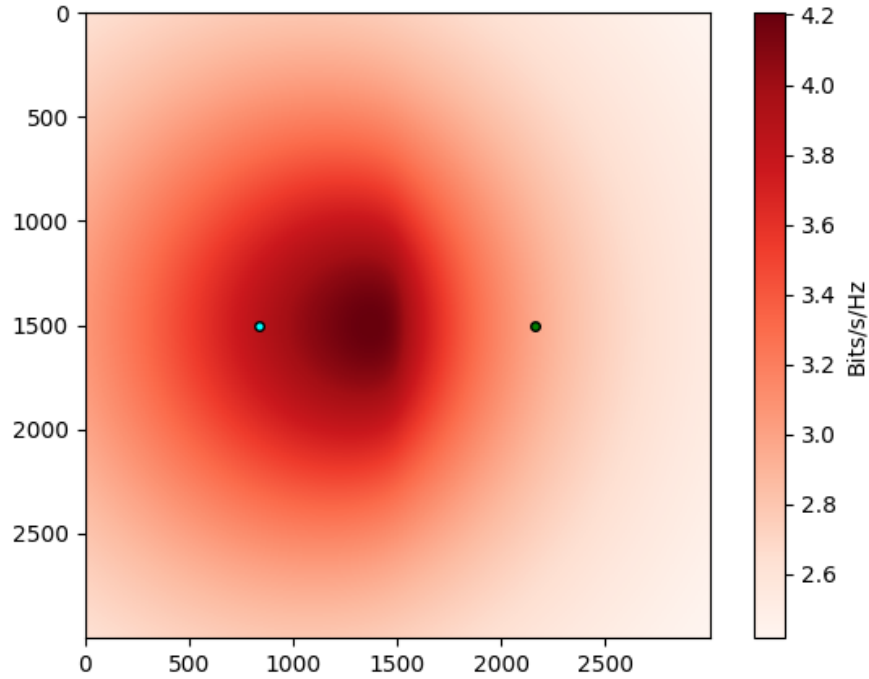


Figure 4.5: Cut-set upper bound capacity heatmap for different relay placements, with source node (teal) on the left and destination node (green) on the right.

The bits/s/Hz values can be seen on the color gradient bar to the right, or in the visualization created by Python we can hover over an area to get a capacity calculation reading for a specific point.

4.2.1 Real-World Heatmap

To make a more realistic scenario for placing a relay, the heatmap was overlaid on top of the measuring location. Figure 4.6 gives us an indication of where to place a cooperative relay. To improve the throughput between our source and destination nodes, we should place our node somewhere in the dark red area.

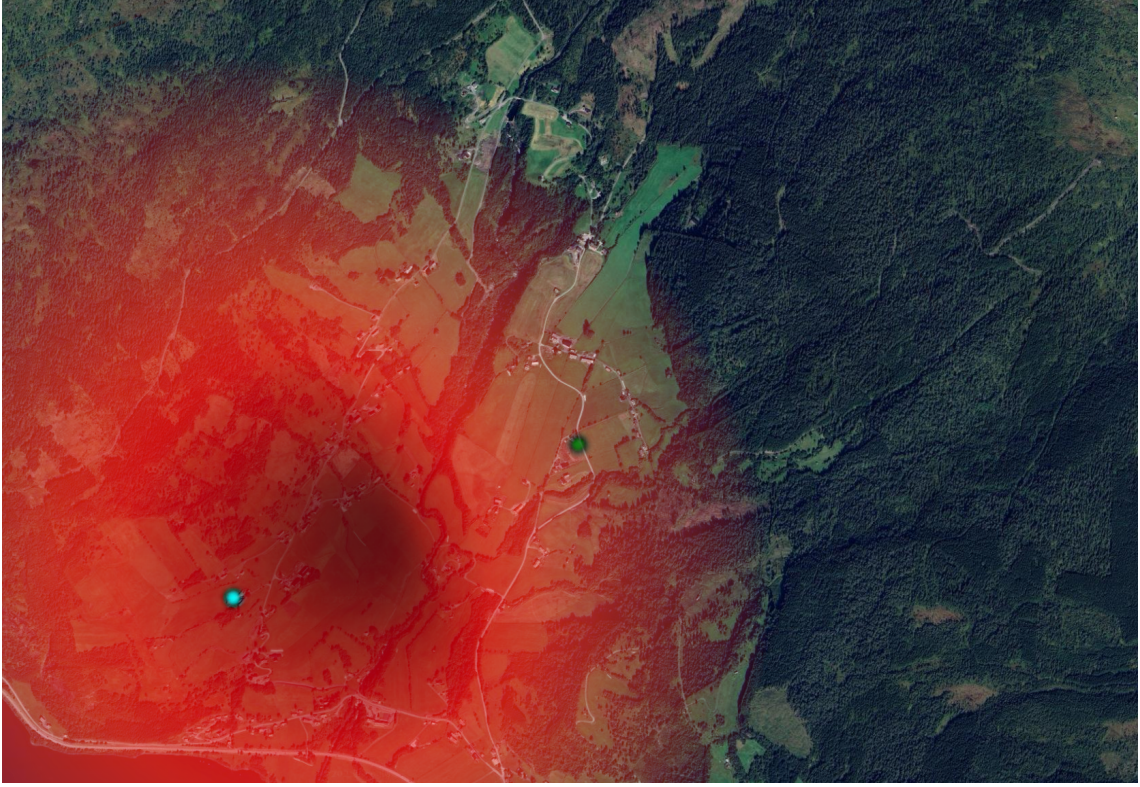


Figure 4.6: Capacity heatmap on top of a real world map. Source node in teal, and destination node in green.

4.2.2 Scheme Specific Heatmaps

Changing the code slightly introduces relay specific heatmaps for the known scheme capacity formulas. The upper bound for amplify-forward can be seen in Figure 4.7a, while lower bound calculations for compress-forward and decode-forward can be seen in Figure 4.7c and Figure 4.7b.

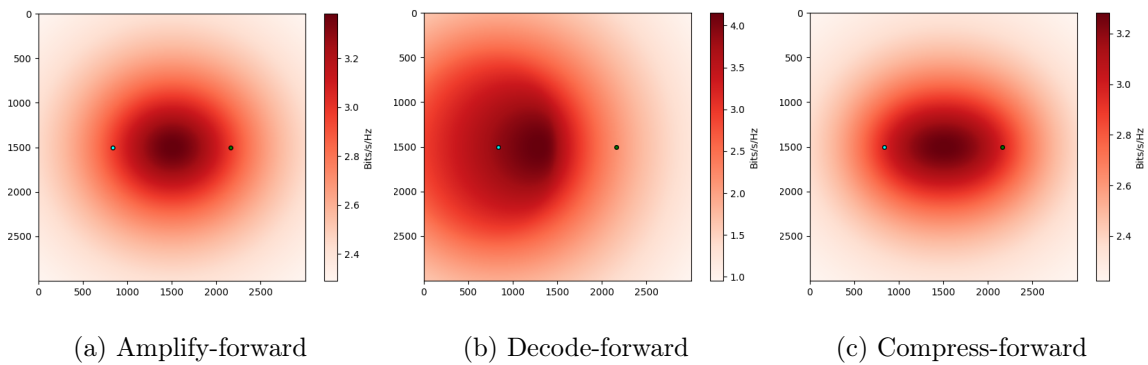


Figure 4.7: Bits/s/Hz capacity heatmaps for different relay schemes. Source node (teal) is on the left; destination node (green) is on the right.

4.3 Simulation and Implementation in GNU Radio

For simulations I landed at the model shown in Figure 4.8. This model assumes all proper signal processing is done at the simulated blocks, and is used as a guide to import empirical data to simulate path loss and noise, as well as how to simulate the effects of fading and timing differences.

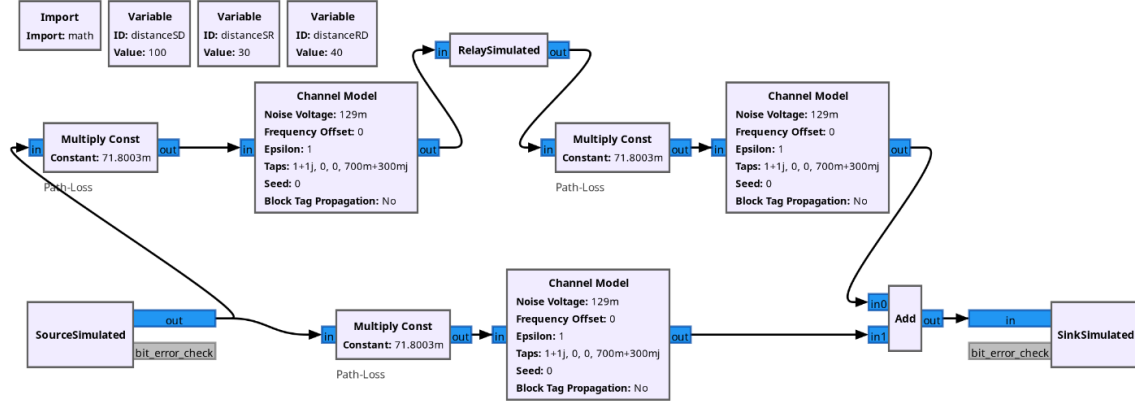


Figure 4.8: Single-relay channel with noise, fading and path loss in GNU Radio companion.

Using this simplified model I showed how easily it can be converted to an implementation using existing software-defined-radio blocks in GNU Radio Companion. The source node (Figure 4.9), destination (Figure 4.10), and relay (Figure 4.11), displays the simplicity of this conversion.

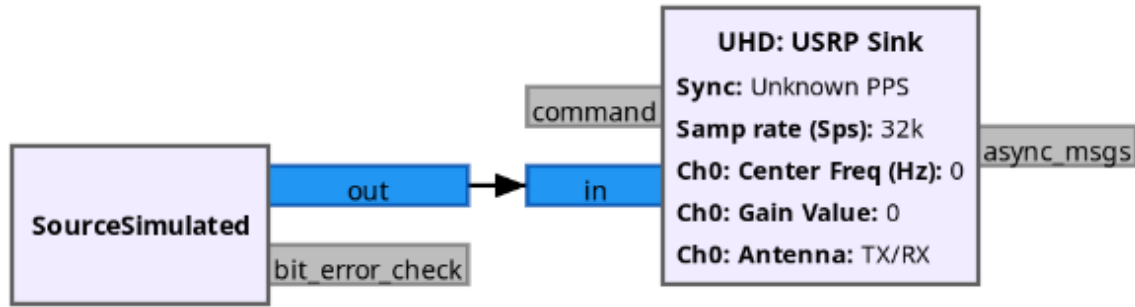


Figure 4.9: Flowgraph of SDR implementation for a source relay node.

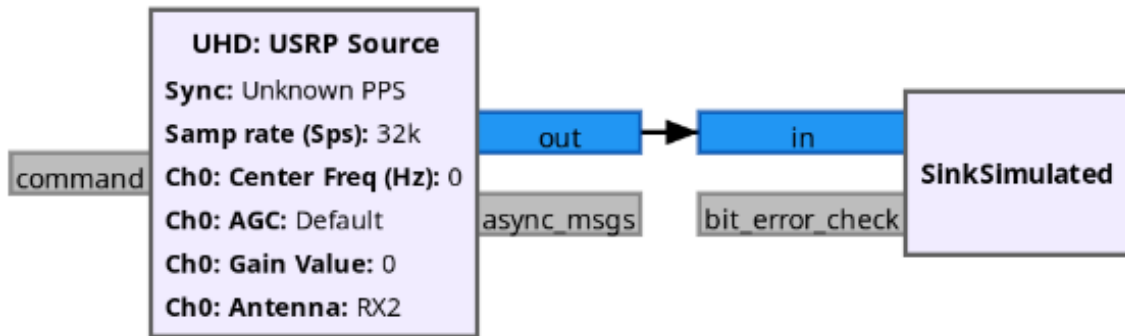


Figure 4.10: Flowgraph of SDR implementation for a destination node.



Figure 4.11: Flowgraph of SDR implementation for a relay node.

The flowgraph in Figure 4.12 shows the correct implementation of a source node. In this, the custom block labeled "Source" only does data generation and error encoding, while GNU Radio in-tree module blocks are used for signal processing.

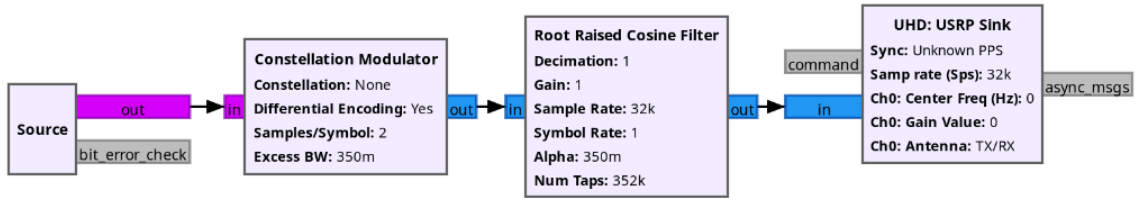


Figure 4.12: Flowgraph of SDR implementation for a source node with GNU Radio modulation and pulse shaping.

The transmitter side of a wireless channel can often be implemented with only modulation and pulse shaping (root raised cosine filter in Figure 4.12). However, while the implementation of the source node is often as simple as modulation and pulse shaping, the broad range of demodulation, synchronization and other architectural choices at the other nodes make these a far more complex endeavor.

I have included the code for the GNU Radio custom source blocks. The complex output block labeled "SimulatedSource" and the byte output block labeled "Source". These were made as examples and to show the nature of how to input and output data using custom blocks.

- The complex output block shows an example using the message passing system for error checking, and a simple BPSK modulation scheme. Found in appendix 5.2 C++ and 5.2 YAML
- The byte output block example has a simple repeating code implementation. Found in appendix 5.2 C++ and 5.2 YAML

Although no working simulation or implementation was made in section 3.3, I have shown how to use empirical data for simulations in GNU Radio, and done the groundwork for implementing both simulations and implementations of a wireless relay channel. The high complexity and many architectural choices in these systems strengthen my belief that there are discrepancies in throughput estimates derived from formulas, simple simulations, when compared to actual achievable real-world throughput.

Chapter 5

Discussion, Further Work and Conclusion

5.1 Discussion and Further work

5.1.1 Data Collection

While trying to make the measurements as simple as possible and only measuring from line-of-sight locations, we can see in Figure 4.4 that there are differences from the free path model (2.3.4). This indicates that along with our measurements we also capture some multiple-paths.

While my measurements were done in a rural area with little to no interference from other RF equipment, a more urban measuring location might have vastly different noise characteristics. For urban areas we might also want to use the concept of signal-to-interference-plus-noise instead of only signal-to-noise ratio.

As explained in section 2.2.1 a spectrum analyzer displays the magnitude of a signal as a function of frequency. While this is good for measuring the presence and the strength of a signal, it tells us nothing about the quality or distortion of the received signal. A software-defined radio on the other hand can separate, display and process all components of the signal individually.

There are built in blocks in GNU Radio for visualizing and analyzing a signal in real time.

- **QT GUI Time Sink:** Displays the signals amplitude as a function of time (similar to an oscilloscope). We can use this for checking timing or if a signal is distorted.
- **QT GUI Frequency Sink:** Displays the signal like a spectrum analyzer does.
- **QT GUI Constellation Sink:** Plots constellation points from samples of the signal.

By transmitting a stable sine wave (or similar) and implemented some kind of synchronization between sender and receiver, I could have received the signal and displayed it using the frequency sink of a software-defined radio. This would have given

a chance to analyze for multi-path propagation and look for distortions and timing difference of the received signal.

Given more time for the thesis I would most definitively touched on these visualizations as they are paramount in finding errors and analyzing a received or simulated signal. By using a File Sink block in GNU Radio we could have also saved the signal to a file for later analysis, again proving the flexibility of GNU Radio.

While creating methods for capturing all the effects of a wireless channel is far from impossible, the work needed to measure and fit the complex data to a simulated model might be more than implementing and testing a solution instead.

Although the software-defined radio seems superior to the spectrum analyzer in this context and is most definitively a more flexible tool, a spectrum analyzer is regarded to be more precise in measuring signal strength.

The data comparison in Section 4.1.2 shows how much impact the simulation of modulation and noise has on the throughput, compared to the theoretical results. Although the experimental data was imprecise, we can clearly see yet another loss of throughput from simulation to the testbed implementation by comparing Figure 4.2 and Figure 4.3.

5.1.2 Relay Capacity Visualization

Figure 4.7 shows the capacity heatmaps of the three cooperative relay schemes amplify-forward, decode-forward, and compress-forward. The code and visualization in its current state might seem novel, but regardless of the calculated capacity at each relay location the characteristics of the heatmap could help us make an educated decision of relay scheme and node location based off of the heatmaps. For improving already existing networks with set in stone nodes, a similar approach could be useful for scheme decisions within larger networks.

A rather easy improvement for the map visualization in Figure 4.6 could have been to use elevation map data to figure out the real distances between the nodes.

Another use case for the heatmap is to use oneself as a "moving antenna" and automate the logging of measurements and position to enter into a similar heatmap. This could be done with simple spectrum analyzer measurements, but also used with an implementation on an SDR to get precise implementation specific throughput readings.

Finally using the *coordinated_relay_capacity* function on line 58 in 5.2 we could have have used real GPS coordinates to calculate the capacity of three existing antennas. To do this we would have to normalize the coordinates of three antennas to (0,0) for the source, (0,"S-D distance") for the destination and, (x, y) for the relay.

5.1.3 GNU Radio and Software-Defined Radios

I was hoping to gain some experimental results from the simulation and implementation made in section 3.3, then compare them to capacity calculations from the literature. However, the complexity of successfully transmitting a wireless signal efficiently proved challenging. I instead hope to have done the groundwork in firstly showing how to simulate natural effects from empirical data, and with the correct

work-flow be able to easily convert these simulations into testbed implementations.

A simpler alternative route could have been to transmit bits directly throughout the GNU Radio flowgraph. However, the hard part in this would have been simulating the physical effects of wireless channels on bits and confidently assessing the correctness of the results gained from it.

While my flowgraphs leave much for the reader to implement, the different relaying schemes often call for specific modulation and synchronization. I think the example of the simulated flowgraph in Figure 4.8 is a good starting point and might be the easiest option for converting an existing programmatic simulation into GNU Radio. Examples of implementing simple modulation and synchronization can be found in the GNU Radio Wiki [9].

While I discourage the simplification of simulations we might want to do this in the development phase to find errors or to test specific parts of the system. To simplify the superimposing of signals we might for instance create several input blocks on a receiver block to perfectly separate two received signals in code.

Simulation of cross-channel interference is simply a new unwanted wireless channel being added on top of a received signal using the add block. We can model this by using the single-channel model in Figure 3.11 and adding it to our wanted signal using the add block. For larger networks this will make our flowgraph very crowded and we might wish we had gone for a testbed implementation instead.

5.2 Conclusion

While the tools and measuring equipment presented were useful for gathering data that could be used in theoretical calculations for channel capacity, I have found that the simple signal strength and noise floor combination to be lacking when trying to simulate the complexity of wireless channels.

The code created for visualizing capacity of different relay placements can be used mostly for conceptualizing the different relay schemes, and needs improvements to be useful in a real scenario. These calculations use the empirical data captured and therefore will include the same discrepancies.

I did not reach my ambitious goal of creating a working simulation and implementation. However, I believe to have started the groundwork for this and proven how easy it can be to convert simulations in GNU Radio to working physical implementations.

While the data from the experimental study I found were imprecise, we could see that by only simulating modulation and noise we had a significant drop in throughput compared to the theoretical lower-bounds. With the high complexity and many architectural choices in the signal processing of GNU Radio I hypothesize that there is a huge discrepancy between the throughput estimates derived from formulas and simplified simulations.

Bibliography

- [1] G. Maps. (2025) [map of voss municipality].
- [2] T. M. Cover and A. A. El Gamal, “Capacity theorems for the relay channel,” *IEEE Transactions on Information Theory*, 1979.
- [3] C. E. Shannon, “A mathematical theory of communication,” *Bell System Technical Journal*, vol. 27, no. 3, July 1948.
- [4] L. R. Ford and D. R. Fulkerson, *Flows in Networks*. Princeton, NJ: Princeton University Press, 1962.
- [5] A. E. Gamal and Y.-H. Kim, *Network Information Theory*. Cambridge University Press, 2011.
- [6] H. H. Sneessens, J. Louveaux, and L. Vandendorpe, “Turbo-coded decode-and-forward strategy resilient to relay errors,” in *2008 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2008.
- [7] K. Q. Huynh and T. Aulin, “Improved Iterative Decoders for Turbo-Coded Decode-and-Forward Relay Channels,” in *2012 IEEE Vehicular Technology Conference (VTC Fall)*. IEEE, 2012.
- [8] A. S. Avestimehr, S. N. Diggavi, and D. N. C. Tse, “Wireless network information flow: a deterministic approach,” *IEEE Transactions on Information Theory*, 2011.
- [9] G. R. Foundation, “Gnu radio documentation and manual,” <https://wiki.gnuradio.org/index.php>, 2025, accessed: August 15, 2025.
- [10] —, “Gnu radio selected papers,” https://wiki.gnuradio.org/index.php/Selected_Academic_Papers, 2025, accessed: November 20, 2025.
- [11] B. A. Channa, M. Khan, M. Rehman, and S. Tanoli, “Ni-usrp based experimental evaluation of cooperative communication over direct communication in lab environment,” pp. 1–5, 01 2018.
- [12] M. Duarte, A. Sengupta, S. Brahma, C. Fragouli, and S. Diggavi, “Quantize map forward (qmf) relaying: An experimental study,” in *Proceedings of the Fourteenth ACM International Symposium on Mobile Ad Hoc Networking and Computing (MobiHoc)*. ACM, 2013.
- [13] tinySA. tinysa | homepage. <https://tinysa.org/>. tinySA project. Accessed: August 20, 2025.

- [14] *Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications*, Institute of Electrical and Electronics Engineers Std., 1997.
- [15] WARP Project, “WARPLab Wiki,” Wiki page available at <https://warpproject.org/trac/wiki/WARPLab>, accessed: November 20, 2025.
- [16] V. Nagpal, I.-h. Wang, M. Jorgovanovic, D. Tse, and B. Nikolic, “Coding and system design for quantize-map-and-forward relaying,” *IEEE Journal on Selected Areas in Communications*, vol. 31, no. 8, 2013.

Appendix: Program listing

Python WiFi scanner:

```
1 import time
2 import subprocess
3 import os
4 import sys
5 import re
6 import datetime
7 from collections import defaultdict
8
9 def check_root_privileges():
10     if not os.geteuid() == 0:
11         print("This script needs to be run with sudo/root
12             privileges for network interface manipulation.")
13         sys.exit(1)
14
15 def choose_interface():
16     try:
17         # The 'iw dev' command lists all wireless devices
18         stdout, _ = run_command(["iw", "dev"],
19             capture_output=True)
20         interfaces = []
21         for line in stdout.splitlines():
22             if "Interface" in line:
23                 interfaces.append(line.strip().split()[1])
24     except Exception as e:
25         print(f"Could not get interfaces: {e}")
26         return None
27
28     print("\nAvailable wireless interfaces:")
29     for i, interface in enumerate(interfaces, 1):
30         print(f" {i}) {interface}")
31
32     while True:
33         try:
34             choice = input(f"Please select an interface
35                 (1-{len(interfaces)}): ")
36             index = int(choice) - 1
37             if 0 <= index < len(interfaces):
38                 interfaceName = interfaces[index]
39                 print(f"Interface selected: {interfaceName}")
40                 break
41             else:
42                 print("Invalid selection. Please try again.")
43         except ValueError:
44             print("Invalid input. Please enter a number.")
45     return interfaceName
```

```

44 def run_command(command, check_error=True,
45 capture_output=False, shell=False):
46     """Helper function to run shell commands."""
47     print(f"Executing: {' '.join(command) if not shell else
48 command}")
49     try:
50         process = subprocess.run(command, check=check_error,
51 capture_output=capture_output, text=True, shell=shell)
52         if capture_output:
53             return process.stdout.strip(),
54 process.stderr.strip()
55         return True
56     except subprocess.CalledProcessError as e:
57         print(f"Error executing command: {e}")
58         if capture_output:
59             print(f"STDOUT: {e.stdout}")
60             print(f"STDERR: {e.stderr}")
61         return False
62     except FileNotFoundError:
63         print(f"Error: Command '{command[0]}' not found. Make
64 sure it's installed and in your PATH.")
65         return False
66     except Exception as e:
67         print(f"An unexpected error occurred while running
68 command: {e}")
69         return False
70
71 def get_wifi_info(interface):
72     data, _ = run_command(f'sudo iw dev {interface} scan
73 2>/dev/null', check_error=False, capture_output=True,
74 shell=True)
75
76     if not data:
77         print("Could not retrieve wireless scan data. Check
78 interface and permissions.")
79         return defaultdict(list)
80
81     networks =
82 re.split(r'BSS\s+([0-9a-fA-F:]{17})(?:\s+(on\s+[^)]+))?\s+',
83 data)[1:]
84     wifi_data = defaultdict(list)
85
86     #Iterate through the chunks of data for each BSS.
87     for i in range(0, len(networks), 2):
88         mac_address = networks[i]
89         ap_data = networks[i+1]
90
91         #Temporary dictionary to store the parsed data for the
92 current AP.
93         current_ap_info = {
94             "ssid": "Hidden Network",

```

```

83         "rssi": None,
84         "channel": None,
85         "frequency": None,
86         "security": "Open",
87         "protocol": "802.11b/g"
88     }
89
90     #SSID
91     ssid_match = re.search(r'SSID: (.+)', ap_data)
92     if ssid_match:
93         current_ap_info["ssid"] =
ssid_match.group(1).strip()
94
95     #Signal strength
96     signal_match = re.search(r'signal: (-?\d+\.\d+)\s+dBm',
ap_data)
97     if signal_match:
98         current_ap_info["rssi"] =
float(signal_match.group(1))
99
100    #Channel
101    channel_match = re.search(r'\*\s+primary
channel:\s+(\d+)', ap_data)
102    if channel_match:
103        current_ap_info["channel"] =
int(channel_match.group(1))
104
105    #Frequency
106    freq_match = re.search(r'^\s+freq:\s+(\d+)', ap_data,
re.MULTILINE)
107    if freq_match:
108        current_ap_info["frequency"] =
int(freq_match.group(1))
109
110    #Determine the highest supported Wi-Fi protocol.
111    if "HE Capabilities" in ap_data:
112        current_ap_info["protocol"] = "802.11ax (Wi-Fi 6)"
113    elif "VHT Capabilities" in ap_data:
114        current_ap_info["protocol"] = "802.11ac (Wi-Fi 5)"
115    elif "HT Capabilities" in ap_data:
116        current_ap_info["protocol"] = "802.11n (Wi-Fi 4)"
117
118    #Determine the security type.
119    rsn_match =
re.search(r'RSN:\s*([\S\s]+?)(?:\n\s*[^W]|$)', ap_data)
120    if rsn_match:
121        rsn_block = rsn_match.group(1)
122        # SAE is a indicator for WPA3
123        if "SAE" in rsn_block:
124            current_ap_info["security"] = "WPA3"
125        else:

```

```

126         current_ap_info["security"] = "WPA2"
127
128     elif "WPA:" in ap_data:
129         current_ap_info["security"] = "WPA"
130
131     elif "WEP" in ap_data and "capability" in ap_data:
132         current_ap_info["security"] = "WEP"
133
134     if current_ap_info["rssi"] is not None:
135
136         wifi_data[current_ap_info["ssid"]].append(current_ap_info)
137
138     return wifi_data
139
140 def setup_monitor_mode(interface):
141     print(f"Setting up monitor mode on {interface}...")
142     run_command(["sudo", "airmon-ng", "check", "kill"])
143     stdout, stderr = run_command(["sudo", "airmon-ng", "start",
144     interface], capture_output=True)
145     if not stdout:
146         print("Failed to start monitor mode with airmon-ng.
147         Check interface name and permissions.")
148         return None
149
150     match = re.search(r'\(monitor mode enabled on (.*)\)\'',
151     stdout)
152     if match:
153         monitor_interface = match.group(1)
154         print(f"Monitor mode enabled on {monitor_interface}")
155         return monitor_interface
156     else:
157         print("Could not determine monitor interface name from
158         airmon-ng output. Check output below:")
159         print(stdout)
160         return None
161
162 def stop_monitor_mode(interface):
163     print(f"Stopping monitor mode on {interface}...")
164     run_command(["sudo", "airmon-ng", "stop", interface])
165     print("Restarting NetworkManager and wpa_supplicant...")
166     run_command(["sudo", "systemctl", "start",
167     "NetworkManager"], check_error=False)
168     run_command(["sudo", "systemctl", "start",
169     "wpa_supplicant"], check_error=False)
170     time.sleep(5)
171
172 def main():
173     check_root_privileges()

```



```

170     interfaceName = choose_interface()
171
172     networks = get_wifi_info(interfaceName)
173
174     if networks:
175         print("\n--- Detected Wi-Fi Networks ---")
176         for ssid, protocols in networks.items():
177             print(f"SSID: {ssid}")
178             for protocol_info in protocols:
179                 print(f"    - Protocol:
180 {protocol_info['protocol']}")
181                 print(f"        RSSI: {protocol_info['rssi']} dBm")
182                 print(f"        Channel:
183 {protocol_info['channel']}")
184                 print(f"        Frequency:
185 {protocol_info['frequency']} MHz")
186                 print(f"        Security:
187 {protocol_info['security']}")
188                 print("-" * 20)
189     else:
190         print("No networks found or scan failed.")
191
192 main()

```

Python relay location:

```
1 import numpy as np
2 from scipy.optimize import curve_fit
3 import matplotlib.pyplot as plt
4 import math
5 from typing import Tuple
6 import matplotlib.colors as mcolors
7
8 #Channel Capacity function:  $C(x) = \log_2(1 + x)$ 
9 def shannon_capacity(SN_ratio):
10     return np.log2(1 + SN_ratio)
11
12 #Calculate the relay capacity, given S/N ratio for links where
13     1 = source, 2 = relay, 3 = destination
14 def cutset_upper_bound(S21: float, S31: float, S32: float):
15     if(S21 >= S32):
16         num = np.pow(np.sqrt(S21*S32) +
17             np.sqrt(S31*(S31+S21-S32)), 2)
18         div = S31 + S21
19         return shannon_capacity(num / div)
20     else:
21         return shannon_capacity(S31 + S21)
22
23 def AF_upper_bound(S21: float, S31: float, S32: float):
24     num = S21 * S32
25     div = S21 + S32 + 1
26     return shannon_capacity(S31 + (num / div))
27
28 def CF_lower_bound(S21: float, S31: float, S32: float):
29     num = S21 * S32
30     div = S31 + S21 + S32 + 1
31     return shannon_capacity(S31 + (num / div))
32
33 def DF_lower_bound(S21: float, S31: float, S32: float):
34     if(S21 >= S31 + S32):
35         num = np.pow(np.sqrt(S31 * (S21-S32)) + np.sqrt(S32 *
36             (S21-S31)), 2)
37         return shannon_capacity(num / S21)
38     else:
39         return shannon_capacity(S21)
40
41 #returns S/N given distance.
42 def power_law(d, B, alpha):
43     return B * (d ** (-alpha))
44
45 Point2D = Tuple[int, int]
46
47 def euclidean_distance(point1: Point2D, point2: Point2D):
48     x1, y1 = point1
```

```

47     x2, y2 = point2
48
49     dx = x2 - x1
50     dy = y2 - y1
51
52     distance_squared = dx**2 + dy**2
53     distance = math.sqrt(distance_squared)
54
55     return distance
56
57 #calculated the cooperative relay capacity given locations of
    nodes and power rule parameters
58 def coordinated_relay_capacity(source: Point2D, relay: Point2D,
    destination: Point2D, B, alpha):
59     dist_21 = euclidean_distance(source, relay)
60     dist_31 = euclidean_distance(source, destination)
61     dist_32 = euclidean_distance(relay, destination)
62
63     #Find S/N given distance
64     S21 = power_law(dist_21, B, alpha)
65     S31 = power_law(dist_31, B, alpha)
66     S32 = power_law(dist_32, B, alpha)
67
68     #cutset upper bound
69     capacity = cutset_upper_bound(S21, S31, S32)
70
71     #amplify forward specific upper bound
72     #capacity = AF_upper_bound(S21, S31, S32)
73
74     #decode forward specific lower bound
75     #capacity = DF_lower_bound(S21, S31, S32)
76
77     #compress forward specific lower bound
78     #capacity = CF_lower_bound(S21, S31, S32)
79
80     return capacity
81
82 #create a heatmap for capacity of relay channel given distance
    between source and destination Todo: big runtime
    optimizations..
83 def relay_placement_heatmap(distance, B, alpha, mapSize):
84     #source -> destination S/N is constant
85     S31 = power_law(distance, B, alpha)
86     direct_capacity = shannon_capacity(S31)
87
88     source_x = round(mapSize/2 - distance/2)
89     source_y = round(mapSize/2)
90     destination_x = round(mapSize/2 + distance/2)
91     destination_y = round(mapSize/2)
92
93     heatmap = np.zeros((mapSize, mapSize))

```

```

94     for y in range(mapSize):
95         for x in range(mapSize):
96             if((x == source_x and y == source_y) or (x ==
destination_x and y == destination_y)):
97                 heatmap[y, x] = np.nan
98                 continue
99
100             #heatmap of capacity
101             heatmap[y, x] =
coordinated_relay_capacity((source_x, source_y), (x, y),
(destination_x, destination_y), B, alpha)
102
103             #heatmap of relay capacity gain when compared to
direct link capacity
104             #heatmap[y, x] =
coordinated_relay_capacity((source_x, source_y), (x, y),
(destination_x, destination_y), B, alpha) - direct_capacity
105
106             #heatmap of capacity to direct link capacity ratio
107             #heatmap[y, x] =
coordinated_relay_capacity((source_x, source_y), (x, y),
(destination_x, destination_y), B, alpha) / direct_capacity
108
109     return heatmap
110
111 #Measured data
112 measured_data = {
113     'Distance': np.array([7.0, 50.0, 240.0, 670.0, 1320.0]),
114     'signalStrength': np.array([-57, -73, -90, -93, -96])
115 }
116
117 noise_floor = -105
118 #calculate SNR
119 measured_SNR = []
120 for signal in measured_data["signalStrength"]:
121     measured_SNR.append(signal - noise_floor)
122
123 #calculate SN ratio
124 SN_ratio = []
125 for SNR in measured_SNR:
126     SN_ratio.append(pow(10, (SNR/10)))
127
128 #curve fitting
129 p0 = [3000000.0, 2.0] #initial guess
130 params, pcov = curve_fit(power_law, measured_data['Distance'],
SN_ratio, p0=p0, maxfev=5000)
131 B, alpha = params
132
133 d_fit = np.linspace(measured_data['Distance'].min(),
measured_data['Distance'].max(), 500)
134 SN_ratio_fit_linear = power_law(d_fit, B, alpha)

```

```

135 SNR_fit_dB = 10 * np.log10(SN_ratio_fit_linear)
136
137 # Plotting: SNR vs. Distance
138 plt.figure(figsize=(10, 6))
139 plt.scatter(
140     measured_data['Distance'],
141     measured_SNR,
142     label='Measured SNR (dB) Data',
143     color='red',
144     marker='o',
145     zorder=5
146 )
147 plt.plot(
148     d_fit,
149     SNR_fit_dB,
150     label=f'Fitted Path Loss Model',
151     color='blue',
152     linestyle='--',
153     linewidth=2,
154     zorder=2
155 )
156 plt.xlabel('Distance ($d$)', fontsize=12)
157 plt.ylabel('Signal-to-Noise Ratio (dB)', fontsize=12)
158 plt.title('SNR vs. Distance: Power Law Model Fit', fontsize=14)
159 plt.legend(fontsize=10)
160 plt.grid(True, which="both", linestyle="--", linewidth=0.5)
161 plt.show()
162
163
164 #Heatmap plot for capacity of cooperative relay channel
165 mapSize = 3000
166 distance = 1320
167
168 heatmap = relay_placement_heatmap(distance, B, alpha, mapSize)
169 data_min = np.nanmin(heatmap)
170 data_max = np.nanmax(heatmap)
171
172 plt.figure(figsize=(7, 5))
173
174 plt.imshow(
175     heatmap,
176     cmap="Reds",
177     interpolation='nearest',
178     vmin=data_min,
179     vmax=data_max
180 )
181 plt.colorbar(label='Bits/s/Hz')
182 source_x = round(mapSize/2 - distance/2)
183 source_y = round(mapSize/2)
184 destination_x = round(mapSize/2 + distance/2)
185 destination_y = round(mapSize/2)

```

```
186 # plot source and destination markers
187 plt.plot(
188     source_x,
189     source_y,
190     marker='o',
191     color='cyan',
192     markersize=4,
193     markeredgecolor='black',
194     label='Source'
195 )
196 plt.plot(
197     destination_x,
198     destination_y,
199     marker='o',
200     color='green',
201     markersize=4,
202     markeredgecolor='black',
203     label='Destination'
204 )
205
206 plt.show()
```

Custom Block: complex

```
1  #include "SourceSimulated_impl.h"
2  #include <gnuradio/io_signature.h>
3  #include <random>
4  #include <gnuradio/logger.h>
5  #include <pmt/pmt.h>
6  #include <vector>
7
8  namespace gr {
9  namespace hansmod {
10
11  using output_type = std::complex<float>;
12  SourceSimulated::sptr SourceSimulated::make() { return gnuradio
    ::make_block_sptr<SourceSimulated_impl>(); }
13
14  /*
15   * The private constructor
16   */
17  SourceSimulated_impl::SourceSimulated_impl()
18      : gr::sync_block("SourceSimulated",
19                      gr::io_signature::make(0, 0, 0),
20                      gr::io_signature::make(
21                          1 /* min outputs */, 1 /*max outputs */,
22                          sizeof(output_type))),
23      d_rng(std::random_device{}())
24  {
25      message_port_register_out(pmt::mp("bit_error_check"));
26  }
27
28  /*
29   * Our virtual destructor.
30   */
31  SourceSimulated_impl::~SourceSimulated_impl() {}
32
33  int SourceSimulated_impl::work(int noutput_items,
34                                gr_vector_const_void_star&
35                                input_items,
36                                gr_vector_void_star& output_items
37                                )
38  {
39      auto out = static_cast<output_type*>(output_items[0]);
40
41      std::uniform_int_distribution<uint8_t> distribution(0, 1);
42      std::vector<uint8_t> generated_bits(noutput_items);
43
44      //Generate random bits
45      for (int i = 0; i < noutput_items; i++) {
46          generated_bits[i] = distribution(d_rng);
47      }
```

```

46 //send the data to the message port for error checking at
    the receiver.
47 pmt::pmt_t message = pmt::init_u8vector(noutput_items,
    generated_bits.data());
48 message_port_pub(pmt::mp("bit_error_check"), message);
49 std::stringstream firstbits;
50 for (size_t i = 0; i < std::min(10, noutput_items); ++i) {
51     firstbits << (int)generated_bits[i] << " ";
52 }
53
54 //todo:
55 //encoding
56
57 //example BPSK modulation
58 for (int i = 0; i < noutput_items; i++) {
59     float real_part = (generated_bits[i] == 0) ? 1.0f : -1.0
        f;
60     out[i] = output_type(real_part, 0.0f);
61 }
62
63 return noutput_items;
64 }
65
66 }
67 }

```


Source block yaml config

```
1 id: hansmod_Source
2 label: Source
3 category: '[hansmod]'
4
5 templates:
6   imports: from gnuradio import hansmod
7   make: hansmod.Source()
8
9 outputs:
10 - domain: stream
11   dtype: byte
12 - domain: message
13   id: bit_error_check
14   optional: true
15
16 file_format: 1
```

Custom block: byte

```
1 #include "Source_impl.h"
2 #include <gnuradio/io_signature.h>
3 #include <random>
4 #include <vector>
5
6 namespace gr {
7 namespace hansmod {
8
9 #pragma message("set the following appropriately and remove
10 this warning")
11 using output_type = uint8_t;
12 Source::sptr Source::make() { return
13     gnuradio::make_block_sptr<Source_impl>(); }
14
15 /*
16  * The private constructor
17  */
18 Source_impl::Source_impl()
19     : gr::sync_block("Source",
20         gr::io_signature::make(0, 0, 0),
21         gr::io_signature::make(
22             1 /* min outputs */, 1 /*max outputs
23             */, sizeof(output_type))),
24         d_rng(std::random_device{}())
25 {
26 }
27
28 /*
29  * Our virtual destructor.
30  */
31 Source_impl::~Source_impl() {}
32
33 int Source_impl::work(int noutput_items,
34     gr_vector_const_void_star& input_items,
35     gr_vector_void_star& output_items)
36 {
37     auto out = static_cast<output_type*>(output_items[0]);
38
39     std::uniform_int_distribution<uint8_t> distribution(0, 1);
40     int out_index = 0;
41
42     for (int i = 0; i < num_input_bits; i++) {
43         uint8_t input_bit = distribution(d_rng);
44
45         out[out_index++] = input_bit;
46         out[out_index++] = input_bit;
47         out[out_index++] = input_bit;
48     }
49 }
```

```
47  
48     return out_index;  
49 }  
50  
51 }  
52 }
```

SourceSim block yaml config

```
1 id: hansmod_SourceSimulated
2 label: SourceSimulated
3 category: '[hansmod]'
4
5 templates:
6   imports: from gnuradio import hansmod
7   make: hansmod.SourceSimulated()
8
9 outputs:
10 - domain: stream
11   dtype: complex
12 - domain: message
13   id: bit_error_check
14   optional: true
15
16 file_format: 1
```