UiA University of Agder

Master

Estimating Drone Count in a Scene using 5G CSI and Multi-modal 1D-CNN

Sriram Mahateja Akella

Sreenivasa Reddy Yeduri
Linga Reddy Cenkeramaddi

University of Agder, 2024
Faculty of Engineering and Science
Department of Engineering and Sciences

Acknowledgements

I would like to express my sincere gratitude to my advisor, Dr. Sreenivas Reddy Yeduri and Prof. Linga reddy Cenkeramaddi, for their guidance and support throughout this research. Special thanks to my colleagues and fellow researchers at the ACPS lab for their collaboration and technical support.

I also extend my appreciation to the members of the gnuradio and python community for their invaluable assistance and insights, which greatly contributed to the success of this research.

Methods and Tools

During the course of this research, various tools and resources were utilized to aid in data collection, analysis, and the writing process. Specifically, ChatGPT, an AI language model developed by OpenAI, was used to assist in drafting and refining sections of this thesis. This tool provided suggestions for structuring content, improving clarity, and enhancing the overall quality of the writing. I have not used ChatGPT to write entire paragraphs or chapters, but only to provide suggestions that I have assessed and processed myself.

Abstract

The proliferation of drones has significantly impacted various sectors, including agriculture, surveillance, and delivery services, presenting both opportunities and challenges. Effective management and monitoring of drone activity are crucial for ensuring safety, security, and compliance with regulations. Traditional methods for drone detection, such as radar and visual surveillance, often face limitations in terms of accuracy, range, and susceptibility to environmental conditions. This thesis explores the integration of 5G Channel State Information (CSI) with advanced machine learning techniques to develop a robust system for estimating drone counts in a given scene.

Leveraging the high resolution and precision of 5G technology, this research utilizes CSI data to detect and identify drones. A Multimodal 1D Convolutional Neural Network (1D-CNN) is employed to analyze the one-dimensional time-series data obtained from CSI measurements. The model is designed to automatically learn and extract features indicative of drone presence and movement, thereby enhancing the accuracy and reliability of the detection system.

The study involves the collection of CSI data from controlled experiments, followed by data preprocessing, feature engineering, and model training. The performance of the developed 1D-CNN model is evaluated using standard metrics, demonstrating its efficacy in accurately estimating drone counts. Additionally, the research addresses the challenges and limitations of using 5G CSI and CNNs for drone detection, providing insights into potential improvements and future research directions.

This thesis contributes to the advancement of wireless sensing technologies and offers practical solutions for effective drone management and surveillance, paving the way for further innovations in this field.

Contents

Acknowledgements **i**

Abstract **ii**

List of Figures **vi**

List of Tables **vii**

1 Introduction **1**

1.1 Background 1

1.2 Problem Statement 2

1.3 Research Methodology 3

1.4 Research Limitations 4

1.5 Organisation 4

1.6 Publications 5

1.7 Thesis Outline 5

2 State-of-the-art **7**

2.1 Theoretical Background 7

2.1.1 Channel State Information (CSI) 7

2.1.2 CSI in Sensing Applications: 8

2.1.3 Convolutional Neural Networks 9

2.2 Literature Review 10

3 System Architecture **12**

3.1 System Requirements 12

3.2 System Overview 13

3.3 Building Blocks 13

3.3.1 Laptops 13

3.3.2 Python 14

3.3.3 Gnuradio 15

3.3.4 Ubuntu 24.02 15

3.3.5 USRP E312 16

3.3.6 UAVs 18

4 **Setup & Testing** **20**
4.1 Installation and Configuration Guide 20
4.1.1 Installation of Ubuntu 24.02 20
4.1.2 Installation of Core Packages (Optional) 21
4.1.3 Installation of GnuRadio 3.11 and UHD 4.6 22
4.1.4 Configuration USRP E312 23
4.2 Testing and Verification 24
4.2.1 Secenario 1: Identifying and verifying the operation of USRP E312 . 24
4.2.2 Secenario 2: Testing Signal Transmission and Reception with USRP Using GNU Radio 24

5 **System Implementation** **27**
5.1 5G System Design 27
5.1.1 OFDM Transmitter 27
5.1.2 OFDM Receiver 30
5.2 Dataset Collection 34
5.3 CNN Model Design 36
5.3.1 Data Loading and Preprocessing 36
5.3.2 CNN Model Architecture 39

6 **Results** **43**
6.1 Training and Validation Loss 43
6.2 Training and Validation Accuracy 44
6.3 Confusion Matrix 44

7 **Discussions** **47**
7.1 Interpretation of Results 47
7.2 Comparison with Existing Studies 48
7.3 Challenges 48

8 **Conclusions** **49**

9 **Additional Work** **50**
9.1 Capturing Received Signal Strength Values 50

Bibliography **52**

A **5G System Code** **57**
A.1 OFDM Transmitter and Receiver 57
A.2 Custom CSI Estimator Block 81

B **Drone** **82**

B.1 Drone Count 82
B.2 UAV Velocity 83

C CNN 86

List of Figures

3.1 Overview of the components. 13
3.2 Lenovo Thinkpad T460s running GNU Radio 14
3.3 Acer Aspire Software and Hardware specifications 15
3.4 USRP E312 17
3.5 VERT400 Antenna [35] 18
3.6 Drones used for the study 19

4.1 Identifying device node attachment point 23
4.2 Testing Transmission and Reception using USRP 25
4.3 Visualisation of received signal 26

5.1 OFDM Transmitter Flowchart 29
5.2 OFDM Receiver Flowchart 31
5.3 Dataset collection system overview 35
5.4 CNN model architecture 39

6.1 Training and Validation Loss 43
6.2 Training and Validation Accuracy 44
6.3 CNN Confusion Matrix 45

9.1 Flowchart of RSSI 51

List of Tables

1.1 Overview of the project roles 5
1.2 Overview of meetings 5

3.1 Overview of drones used for the study 18

4.1 Common identifiers and their corresponding keys for USRP 24

6.1 Overall Accuracy of the CNN Model 45
6.2 Precision, Recall, and F1 Score of the CNN model 46

Listings

4.1 Update and Upgrade Packages . . . 21
4.2 Installing Core packages . . . 22
4.3 Installing GnuRadio and UHD . . . 22
4.4 Identifying USRP . . . 24
4.5 Benchmark Testing . . . 24
5.1 Initializing Dataloader class . . . 37
5.2 Method for reading CSI file . . . 37
5.3 Resnet Code . . . 41
A.1 OFDM Transmitter . . . 57
A.2 OFDM Receiver . . . 68
A.3 CSI Estimation . . . 81
B.1 Code used for collecting drone count dataset . . . 82
B.2 Code for collecting the data for UAV velocity . . . 83
C.1 CNN model code . . . 86

Chapter 1

Introduction

This chapter presents a comprehensive overview, beginning with the background that provides context and sets the stage for the research conducted. It then discusses the motivation behind the study, explaining the driving factors and the significance of the research. The chapter outlines the specific research topic and the primary questions it aims to address, followed by the objectives that define the goals and intended outcomes of the study.

Next, the scope of the research are discussed, setting the context of the study by clearly defining what is focused on and what is not. This ensures clarity regarding the specific areas covered and those excluded from the research. The chapter also lists the articles published related to the research, highlighting contributions to the academic community. Finally, it concludes with an overview of the thesis, summarising the structure and content of the subsequent chapters, providing a road-map for the reader.

1.1 Background

The rapid advancement and increasing adoption of drone technology have brought about significant transformations across various industries, including agriculture, delivery services, surveillance, and entertainment[1]. Drones, also known as unmanned aerial vehicles (UAVs), offer numerous benefits due to their versatility, cost-effectiveness, and ability to access hard-to-reach areas. In agriculture, for instance, drones are used for crop monitoring, spraying pesticides, and mapping fields, which enhances precision farming practices and improves crop yields (Liu et al., 2019). Similarly, in the delivery sector, companies like Amazon and UPS are exploring drone-based delivery systems to expedite the delivery process and reduce logistical costs. [2]

However, the proliferation of drones also presents significant challenges, particularly in terms of airspace management, security, and privacy [3]. With the increasing number of drones in the sky, there is a pressing need for robust systems to monitor and manage drone activity effectively. Traditional methods for detecting and counting drones, such

as radar and visual surveillance, often struggle with limitations in range, accuracy, and susceptibility to environmental conditions (Sundaresan et al., 2017). These limitations underscore the necessity for more advanced and reliable sensing technologies.

5G technology, the fifth generation of mobile networks, has emerged as a promising solution to these challenges. With its superior data rates, ultra-low latency, and massive connectivity, 5G offers significant improvements over previous generations of mobile networks (Wang et al., 2019). One of the critical features of 5G for drone detection is its ability to utilize Channel State Information (CSI). CSI provides detailed information about the propagation environment of wireless signals, which can be exploited to detect and identify objects, including drones, within the coverage area (Chen et al., 2018).

In parallel, advancements in machine learning, particularly Convolutional Neural Networks (CNNs), have demonstrated remarkable success in various pattern recognition and data analysis tasks [4]. CNNs are capable of automatically learning and extracting features from raw data, making them highly effective for analyzing complex datasets. When applied to 5G CSI data, CNNs can enhance the accuracy and reliability of drone detection systems by identifying patterns indicative of drone presence and movement (Li et al., 2021).

Despite the potential benefits, the integration of 5G CSI with CNNs for drone detection remains an underexplored area. This research aims to bridge this gap by developing a robust system for estimating drone counts using 5G CSI and a Multimodal 1D-CNN model. By leveraging the high-resolution data provided by 5G technology and the powerful pattern recognition capabilities of CNNs, this study seeks to address the existing challenges and contribute to the advancement of drone monitoring technologies.

1.2 Problem Statement

The widespread use of drones in sectors such as agriculture, delivery, and surveillance has brought about significant challenges in airspace management, security, and privacy (Zhang et al., 2020; Gonzalez-Ruiz et al., 2016). Traditional methods for drone detection, like radar and visual surveillance, face limitations in range, accuracy, and environmental susceptibility (Sundaresan et al., 2017). These challenges necessitate the development of more advanced and reliable sensing technologies.

5G technology, with its high resolution and precision, offers a promising solution for enhanced drone detection and counting through the use of Channel State Information (CSI), which provides detailed insights into the wireless signal propagation environment (Wang et al., 2019). However, effectively leveraging 5G CSI for drone detection presents technical challenges, including accurately interpreting complex data patterns and differentiating drones from other objects or noise (Chen et al., 2018).

Integrating machine learning techniques, particularly Convolutional Neural Networks (CNNs),

with 5G CSI data can significantly improve the accuracy and reliability of drone detection systems (Li et al., 2021). A Multimodal 1D-CNN model can process and analyze the one-dimensional time-series data from CSI measurements, identifying patterns indicative of drone activity. Despite its potential, there is a lack of comprehensive research combining 5G CSI with advanced machine learning models for drone detection (Huang et al., 2020).

This research aims to address the following problems:

- How can 5G CSI data be effectively utilized to detect and count the number of drones in a given scene?

- How can a Multimodal 1D-CNN model be designed, trained, and validated to ensure accurate and reliable drone count estimation?

- What are the challenges and limitations associated with using 5G CSI and CNNs for drone detection, and how can they be mitigated?

By addressing these questions, this research seeks to develop a robust and efficient system for estimating drone counts using 5G CSI and a Multimodal 1D-CNN model.

1.3 Research Methodology

In this study, the general theory needs to be examined, followed by a comprehensive literature review to survey existing methodologies, technologies, and models related to wireless sensing, 5G technology, and drone detection. This review will identify the current state of knowledge, highlight gaps in the existing literature, and provide a foundation for the development of the proposed system.

This research involves the collection of 5G CSI data from a controlled experimental setup. The drones will be flown, and data will be captured using USRPs. The study will include the design and configuration of both hardware and software components. This includes setting up 5G system, configuring drones, and integrating these components with data collection and processing modules. Moreover, the raw 5G CSI data will be preprocessed to remove noise and enhance signal quality for accurate drone count estimation.

The core of the research involves developing a Multimodal 1D-CNN model. The model will be designed to process the preprocessed 5G CSI data and other relevant features to estimate the number of drones in a given scene. The 1D-CNN model will be trained using the collected and preprocessed data. The model will be validated to evaluate its accuracy and robustness.

1.4 Research Limitations

The study aims to develop an accurate system for estimating drone count in a scene using 5G Channel State Information (CSI) and a Multimodal 1D Convolutional Neural Network (1D-CNN), several limitations must be acknowledged that could impact the study's outcomes and applicability.

Firstly, the quality and reliability of the 5G CSI data are dependent upon the performance and stability of the 5G network infrastructure. Variations in signal strength, interference from other devices, and environmental factors such as physical obstructions can affect the accuracy of the CSI data collected. These factors could introduce noise and variability into the dataset, potentially impacting the performance of the 1D-CNN model.

Secondly, the experimental setup is limited by the specific hardware and software configurations used in this study. Additionally, the computational resources required for training and validating the 1D-CNN model might limit the complexity and size of the neural network, potentially affecting its accuracy and robustness.

Thirdly, the study's focus on a particular scene or environment may limit the applicability of the findings to other contexts. The characteristics of the testing environment, such as the size of the area, the number and types of drones, and the presence of other objects or obstacles, may influence the system's performance. Consequently, the results obtained in this specific context may not directly translate to different scenarios or environments without further adaptation and testing.

Moreover, the scope of the study is restricted to the use of 5G CSI and a 1D-CNN model for drone count estimation. While this approach leverages the advantages of 5G technology and deep learning, it may not fully capture the complexities of multimodal data integration or the potential benefits of alternative machine learning models. Exploring other modalities and models could provide additional insights and improve the system's overall performance.

Finally, the time constraints and available resources for this research impose practical limitations on the extent and depth of the study. Comprehensive testing and validation across diverse scenarios and conditions are ideal but may not be feasible within the project's timeframe. As such, the findings and conclusions drawn from this research should be viewed as preliminary and subject to further validation and refinement in future work.

1.5 Organisation

Weekly meetings were held with supervisor to help in consistently tracking progress, opportunity to review, identify deviations, and make necessary adjustments. Receiving regular feedback was provided to improve the quality of work. The meetings were held physicals on campus and occasionally online. As the deadline was approaching, irregular

meetings were conducted to discuss challenges and issues arised from the study. Tabel 1.1 presents the people and their roles for the study. Tabel 1.2 presents an overview of the meetings occured for the study.

Name	Role	Company	Email
Sriram M. Akella	Student	UiA	mahata16@uia.no
Linga R. Cenkeramaddi	Supervisor	UiA	linga.cenkeramaddi@uia.no
Sreenivasa R. Yeduri	Co-Supervisor	UiA	sreenivasa.r.yeduri@uia.no

Table 1.1: Overview of the project roles

Meeting place	Participants	Frequency	Day of week
Online	Student and supervisor	Weekly	Thursday
Physical	Student and supervisor	Irregular	Any weekday

Table 1.2: Overview of meetings

1.6 Publications

This research aims to make contributions to the field of wireless sensing using 5G technology and machine learning. The study's findings and data will be disseminated through multiple publications.

Firstly, two datasets generated during the course of this research will be published. These datasets will include detailed 5G Channel State Information (CSI) captured from the experimental setup and the other will include detailed 5G Received Signal Strength (RSS) values. In addition to the dataset articles, the research findings and methodologies will be documented in a scientific journal article.

This journal publication will detail the development and evaluation of the proposed system for estimating drone counts using 5G CSI and a Multimodal 1D Convolutional Neural Network (1D-CNN). The article will cover the experimental setup, data collection processes, machine learning model development, and the results of our validation studies.

1.7 Thesis Outline

Overview of the structure of the thesis, including chapters and sub-chapters Brief summary of what each section will cover

- **Chapter 2** is dedicated to the theoretical background and a thorough review of the existing literature. The methodologies and theoretical frameworks relevant to the research are discussed, providing a foundation for the study.
- **Chapter 3** focuses on the setup of the system used in the research. The chapter presents a detailed descriptions of all hardware and software components utilised

in the study. The integration of hardware and software is discussed, detailing how the different components work together as a cohesive system and concludes with a discussion of the testing and validation procedures used to ensure the system's functionality and reliability.

- **Chapter 4** presents a comprehensive guide to the software installation process, the configuration of both hardware and software.
- **Chapter 5** provides a detailed account of the data collection and analysis processes. The methods used for data collection are described, covering the techniques and tools employed. The chapter focuses on the development of the machine learning model. The selection of the machine learning approach is justified, explaining the choice of algorithms. The feature selection and engineering process is described, followed by the training and validation of the model.
- **Chapter 6** results of the research are presented and analyzed. The results obtained from the experiments are presented in detail, followed by an analysis and interpretation of these results. The chapter compares the findings with the initial research objectives, highlighting how the results address the research problem. A summary of the key findings is provided, emphasizing the contributions of the research.
- **Chapter 7** provides a comprehensive discussion of the research findings. It begins with an integration of the findings from all chapters, providing a clear view of the research outcomes. The implications of the research for are discussed, highlighting its potential impact. A comparison with previous studies is provided, situating the research within the broader literature.
- **Chapter 8** summarises the main findings of the research, discussing their contributions to knowledge and practical implications. It provides concluding thoughts and reflections on the research journey. The chapter emphasises the significance of the research outcomes and suggests potential areas for future investigation.
- **Chapter 9** discusses additional work that extends beyond the primary research findings. This chapter includes supplementary experiments conducted to support or enhance the main research. It provides an opportunity to present related work that may not fit directly into the core chapters but is valuable for the overall understanding and context of the study.

Chapter 2

State-of-the-art

2.1 Theoretical Background

2.1.1 Channel State Information (CSI)

Channel State Information (CSI) is a critical component in the realm of wireless sensing due to its fine subcarrier-level granularity and its accessibility through commodity Wi-Fi devices. The chapter titled "Understanding of Channel State Information" by Liu et al. (2021) provides a comprehensive introduction to CSI, elaborating on its fundamental aspects and practical applications in wireless sensing technologies [5].

CSI refers to the detailed information about the state of a wireless communication channel, which includes various signal features such as amplitude and phase information of the subcarriers. This data is crucial for understanding and predicting the behavior of the wireless channel, thereby enabling the development of more sophisticated and effective wireless sensing algorithms.

CSI can be represented mathematically as a matrix that describes the effect of the wireless channel on the transmitted signals. Let $\mathbf{H}$ denote the CSI matrix, where $\mathbf{H}$ consists of complex values representing the channel's amplitude and phase response for each subcarrier. The relationship between the transmitted signal $\mathbf{X}$ and the received signal $\mathbf{Y}$ can be expressed as:

$$\mathbf{Y} = \mathbf{H}\mathbf{X} + \mathbf{N} \tag{2.1}$$

where: - $\mathbf{Y}$ is the received signal vector, - $\mathbf{X}$ is the transmitted signal vector, - $\mathbf{N}$ is the noise vector.

Each element h_{ij} of the CSI matrix $\mathbf{H}$ can be decomposed into its amplitude $|h_{ij}|$ and phase $\angle h_{ij}$ components:

$$h_{ij} = |h_{ij}|e^{j\angle h_{ij}} \tag{2.2}$$

The article [5] emphasize the importance of comprehending the underlying model of CSI to leverage its full potential. The chapter breaks down the main components of CSI, including:

Signal Features: Detailed characteristics of the transmitted and received signals, including amplitude, phase, and frequency responses of the subcarriers.

Error Terms: Various sources of errors that can affect the accuracy of CSI measurements, such as noise, interference, and hardware imperfections.

Understanding CSI and its components allows researchers and engineers to develop advanced wireless sensing algorithms. These algorithms can be used in a variety of applications, including indoor positioning, motion detection, and environmental monitoring. The chapter also lists devices that support CSI collection, highlighting the practical feasibility of implementing CSI-based solutions using off-the-shelf hardware.

The work by Liu et al. (2021)[5] serves as a foundational reference for those looking to delve into the field of wireless sensing using CSI. By providing a detailed breakdown of CSI's components and their implications, the chapter aids in the development of robust and efficient wireless sensing technologies.

2.1.2 CSI in Sensing Applications:

In the context of drone detection, CSI can provide valuable information about the presence and movement of drones by analyzing the changes in the wireless channel caused by the drones' movements. The high resolution and precision of 5G CSI enable the detection of subtle changes in the channel, which are indicative of the drones' presence and activity.

- Gesture Recognition: Using CSI data, systems can recognize different gestures by analyzing the variations in wireless signals caused by the movement of the human body [6].
- Vital Sign Monitoring: CSI-based systems can monitor vital signs like respiration and heartbeat by detecting subtle changes in the wireless signal patterns [7].
- Indoor Positioning: CSI data enables precise indoor positioning by analyzing the multipath propagation effects of wireless signals within an environment [8].
- Activity Recognition: By processing CSI data with machine learning algorithms, it is possible to recognize various human activities, enhancing the capabilities of smart home and surveillance systems [9].

2.1.3 Convolutional Neural Networks

Convolutional Neural Networks (CNNs) are a class of deep learning models designed to process data with a grid-like topology, such as images. The fundamental building block of a CNN is the convolutional layer, which applies convolution operations to the input data using a set of filters or kernels. This operation helps in capturing spatial hierarchies by learning features from small, localized regions of the input [10].

The architecture of a typical CNN includes multiple layers such as convolutional layers, pooling layers, and fully connected layers. The convolutional layers perform the core operation of the CNN, extracting features from the input data. Pooling layers, such as max pooling or average pooling, reduce the spatial dimensions of the feature maps, leading to a reduction in the number of parameters and computational complexity. Fully connected layers, placed at the end of the network, perform the classification based on the features extracted by the convolutional layers [10].

Residual Networks (ResNet)

Residual Networks (ResNet) are an advanced form of CNNs that address the problem of vanishing gradients, which hinders the training of very deep networks. ResNet introduces the concept of residual learning, where the network learns residual functions with reference to the layer inputs, rather than learning unreferenced functions directly [11]. The core idea is to use shortcut connections, or skip connections, that bypass one or more layers. These connections enable the gradient to flow directly through the network, allowing for the training of much deeper networks.

A ResNet model consists of residual blocks, each containing two or more convolutional layers with batch normalization and ReLU activation. The shortcut connection adds the input of the block to the output, forming the residual function. This structure helps in mitigating the degradation problem where adding more layers to a deep network leads to higher training error [11].

The ResNet architecture has several variants, including ResNet-18, ResNet-34, ResNet-50, and ResNet-101, each differing in the number of layers and the depth of the network. ResNet-50, for instance, uses a combination of 1x1, 3x3, and 1x1 convolutions in its residual blocks to enhance feature extraction capabilities while maintaining a manageable number of parameters [10].

Applications and Advances

The combination of CNNs and ResNet architectures has led to significant advancements in various fields, including image classification, object detection, and medical image analysis. For example, the use of ResNet-50 in complex image demarcation tasks demonstrates its ability to handle detailed and intricate features in images, providing high accuracy and

robustness in classification tasks [11].

Recent research has also focused on optimizing CNN and ResNet architectures for edge-IoT devices, aiming to reduce energy consumption and computational requirements while maintaining high performance. Techniques such as dependency graph-based pruning and reinforcement learning-driven optimization have been proposed to enhance the efficiency of these models in resource-constrained environments [12].

The theoretical foundations of CNNs and ResNet architectures form the basis for many state-of-the-art deep learning applications. By leveraging convolutional operations and residual learning, these models have achieved remarkable success in various domains. Ongoing research continues to explore new ways to optimize and extend these architectures, ensuring their relevance and effectiveness in emerging applications.

2.2 Literature Review

Recent advancements in wireless sensing technologies have significantly contributed to human activity recognition. In this article [13], the authors discuss the utilization of wireless signal variations caused by human body movements to enable applications such as intrusion detection, daily activity recognition, gesture recognition, vital signs monitoring, and user identification. Krovvidi (2024) explores the fusion of RF and sensor data to enhance activity detection, demonstrating how combining multiple data sources can improve the accuracy and reliability of recognizing human activities [14]. Candell et al. (2024) address the deployment challenges of wireless sensor networks in construction sites, proposing a 5G strategy to enhance activity monitoring and improve site management through real-time data collection and analysis [15]. Yadav et al. (2024) present tinyRadar, a real-time multi-target activity recognition system using LSTM models for edge computing environments, which allows for efficient processing and immediate response to detected activities [16]. Tian et al. (2024) introduce a low-power wearable sensor designed for fall detection in the elderly, leveraging shallow-learning algorithms to provide accurate and timely alerts, thereby enhancing safety for at-risk populations [17]. Mazzieri et al. (2024) examine attention-refined unrolling techniques for sparse sequential micro-Doppler signal reconstruction, highlighting advancements in processing complex signal data for more precise human activity recognition [18]. These studies collectively underscore the potential and ongoing innovations in wireless sensing for human activity recognition, offering insights into various methodologies and applications that drive forward the field's development.

Channel State Information (CSI) has become a focal point in wireless sensing research due to its ability to provide detailed subcarrier-level data, enabling sophisticated analysis and applications. Li et al. [6] investigated the use of WiFi CSI data for gesture recognition, focusing on techniques to enhance data quality and improve recognition accuracy. Their study demonstrated that by applying data enhancement methods, the reliability of gesture

recognition systems could be significantly improved. This research highlights the potential of CSI data in developing robust and accurate gesture recognition systems, which can be applied in various interactive and security applications. In another study, Li et al. [8] developed a method for efficient Doppler speed estimation using WiFi CSI. This approach leverages the fine-grained data provided by CSI to enhance passive tracking capabilities. The proposed method showed that accurate Doppler speed estimation is feasible, which can significantly improve the performance of tracking systems in various environments. This advancement underscores the utility of CSI in developing high-precision tracking systems for applications such as security surveillance and motion detection. Wang [7] explored the application of multi-task contrastive learning to WiFi CSI data for high-accuracy vital sign detection. The study demonstrated that this advanced machine learning technique could be used to monitor vital signs like respiration and heartbeat with high accuracy. This research indicates that CSI data, combined with sophisticated learning algorithms, can be a powerful tool in health monitoring systems, providing non-invasive and continuous monitoring solutions. Mao et al. [9] proposed a novel framework using a modified Generative Adversarial Network (GAN) for cross-domain activity recognition utilizing WiFi CSI. Their framework showcased the flexibility of CSI in diverse application scenarios, demonstrating its capability to adapt to different domains and improve activity recognition performance. This research highlights the potential of CSI in creating adaptable and robust activity recognition systems that can be used in smart homes, workplaces, and public spaces.

1D Convolutional Neural Networks (1D-CNN) are a variant of CNNs specifically designed to process sequential data or time-series data. Unlike 2D-CNNs, which operate on two-dimensional data such as images, 1D-CNNs apply convolution operations along one dimension, making them suitable for tasks involving temporal sequences or signal processing. In wireless sensing, 1D-CNNs are employed to analyze data collected from various sensors, capturing temporal patterns and features that are essential for accurate sensing and recognition tasks [19]. Santhosh et al. (2023) developed an intelligent robust 1D-CNN model for human activity recognition using wearable sensor data. The proposed model, trained on the WISDM dataset, achieved high accuracy in classifying various human activities, demonstrating the effectiveness of 1D-CNN in extracting meaningful features from time-series sensor data [20]. Nguyen et al. (2023) utilized 1D-CNN layers concatenated with a Flatten layer in their low-cost wireless sensor network (LWSN) for monitoring construction emissions. The 1D-CNN effectively processed the sequential data collected from sensors, enabling accurate emission monitoring and environmental assessments [19]. Rostovski et al. (2024) employed a 1D-CNN combined with LSTM for real-time gait anomaly detection using data from wearable sensors. The hybrid model effectively captured both spatial and temporal dependencies in the sensor data, facilitating accurate and real-time detection of gait anomalies [21]. Lau (2024) developed a 1D-CNN model for keystroke inference based on wireless sensing data. The model successfully identified keystroke patterns, highlighting the potential of 1D-CNNs in security and user behavior analysis applications [22].

Chapter 3

System Architecture

This chapter aims to provide a comprehensive overview of the system, highlighting the design choices and considerations that ensure the system meets the research objectives and performance requirements. We begin with the system requirements, highlighting the criteria driving architectural decisions. This is followed by a system overview, providing a perspective on the structure and functionality and an overview of the workflow synthesizes the various components and processes into a cohesive framework.

Next, we examine the building blocks of the system, including hardware like laptops (e.g., Lenovo Thinkpad T460s), and software frameworks such as Python, Gnuradio, and Ubuntu 24.02. The integration of specialized hardware, such as the USRP E312 and Drones used in this study, is also covered, showcasing their roles within the system.

3.1 System Requirements

The requirements involve several aspects that ensure the system's effectiveness and efficiency. The system must be capable of collecting Channel State Information (CSI) from 5G signals and additional sensor data from the environment. This data acquisition should be real-time to facilitate continuous monitoring and analysis. Pre-processing is essential to filter noise and manage missing values, followed by feature extraction and data transformation to prepare the data for analysis.

From a technical perspective, the hardware specifications should include at least 8GB of RAM, multi-core processors, and SSD storage, supported by high-bandwidth network infrastructure for real-time data transfer. The software stack involves using the latest version of a Linux-based operating system, the latest version of Python for programming and machine learning, and an appropriate version of a signal processing tool such as Gnuradio. Operationally, the system requires automated updates and patches to maintain its up-to-date status, along with regular data backups and offsite storage to ensure data integrity.

3.2 System Overview

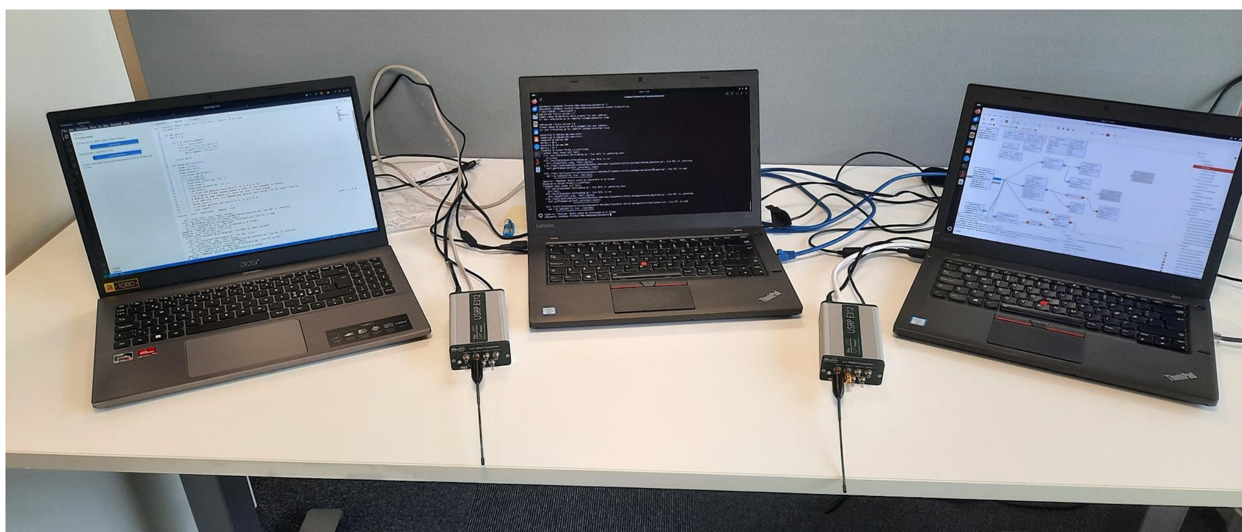


Figure 3.1: Overview of the components.

The core of the system is built around a combination of powerful laptops, versatile software frameworks, and specialized hardware. These components work together to ensure that the system meets the research objectives and performance requirements.

The integration of these components is critical for the system's functionality. Laptops, equipped with the necessary software tools, process and analyze data in real-time. The USRP E312, connected via USB or Ethernet, captures the CSI from 5G signals, which is then processed using GNU Radio running on the laptops. The VERT400 antenna enhances signal reception, ensuring data quality and reliability. Figure 3.1 presents an overview of the components used in the study.

Communication between components is facilitated through well-defined interfaces and protocols, ensuring interoperability and efficient data flow. USB and Ethernet connections enable high-speed data transfer between the USRP E312 and the laptops. Python scripts manage drone communication via libraries such as DroneKit and ROS, supporting autonomous operations and data collection.

3.3 Building Blocks

3.3.1 Laptops

For this study, three laptops were utilised for their portability, computational power, and flexibility. These devices are facilitated for on-site processing, collecting dataset, building machine learning model, and etc, making them a powerful tool for this study's success.

Lenovo Thinkpad T460s

The Lenovo thinkpad T460s is a robust business laptop renowned for its durability, portability, and performance. Equipped with 6th generation Intel Core i5 and up to 8GB of RAM, the T460s provides the necessary computational power to handle intensive signal processing tasks efficiently. Its solid-state drive (SSD) ensures fast data access speeds, which is critical for real-time operations in GNU Radio. [23]

Additionally, the laptop's lightweight and compact design makes it ideal for fieldwork and mobile applications. The T460s also offers multiple USB ports, including USB 3.0, facilitating seamless connections to various software-defined radio (SDR) hardware such as USRP devices. With its durable build quality, the ThinkPad T460s ensures reliability in diverse working conditions, making it a dependable tool for this study. [23] [24]

Figure 3.2: Lenovo Thinkpad T460s running GNU Radio

Acer Aspire 5

Acer Aspire 5 is a highly versatile laptop manufactured in 2020.[25] The laptop running Pop!_OS 22.04 LTS equipped with an AMD Ryzen 7 5825U processor featuring 16 cores and a clock speed of up to 4.546GHz, the laptop offers computational power necessary for developing machine learning model. With 32 GB of RAM, it is capable of handling large datasets and memory-intensive tasks, such as training deep learning models.[25] Figure 3.3 presents further specification of the laptop.

3.3.2 Python

Python is a high-level, interpreted programming language known for its readability and simplicity. Its versatility and power make it crucial to my study due to its wide range of applications and ease of use. Python's extensive standard library and dynamic typing facilitate various tasks, including data analysis, artificial intelligence, scientific computing, and automation. These features make it an ideal choice for complex tasks such as program-

```
OS: Pop!_OS 22.04 LTS x86_64
Host: Aspire A515-47 V1.04
Kernel: 6.8.0-76060800daily20240311-generic
Uptime: 13 hours, 18 mins
Packages: 2661 (dpkg), 18 (flatpak)
Shell: bash 5.1.16
Resolution: 1920x1080, 3440x1440
DE: GNOME 42.9
WM: Mutter
WM Theme: Nordic-bluish-accent
Theme: Catppuccin-Latte-Standard-Maroon-Light [GTK2/3]
Icons: Pop [GTK2/3]
Terminal: x-terminal-emul
CPU: AMD Ryzen 7 5825U with Radeon Graphics (16) @ 4.546GHz
GPU: AMD ATI 04:00.0 Barcelo
Memory: 4098MiB / 31443MiB
```

Figure 3.3: Acer Aspire Software and Hardware specifications

ming drones, developing machine learning models, and transmitting and receiving OFDM signals.

Python's extensive libraries and frameworks, such as DroneKit and ROS (Robot Operating System), facilitate seamless communication with drone hardware and the implementation of autonomous flight algorithms. It's compatibility with various machine learning libraries, including TensorFlow and scikit-learn, enables the efficient development and training of sophisticated models to interpret and analyze data. Additionally, It's robust mathematical and scientific libraries, such as NumPy and SciPy, are instrumental in performing precise and complex calculations necessary for this study.

3.3.3 Gnuradio

According to gnuradio, GNU Radio is a free, open-source toolkit for developing software radios. It offers signal processing blocks and can be used with low-cost RF hardware or in a simulation environment. Widely used in research, industry, academia, government, and by hobbyists, it supports wireless communications research and the development of real-world radio systems.[26]

3.3.4 Ubuntu 24.02

Ubuntu is a free and open-source Linux distribution developed by Canonical Ltd. It is one of the most popular Linux distributions worldwide. It offers regular updates and long-term support (LTS) versions, ensuring stability and security for both personal and enterprise use. Canonical has released Ubuntu 24.04 LTS, codenamed "Noble Numbat," which builds on previous versions and contributions from the open-source community to deliver a secure, optimized platform. [27]

Several features and benefits make Ubuntu an ideal choice for this study.

- Ubuntu offers long-term support versions, such as Ubuntu 24.02 LTS, which provide

five years of security updates and bug fixes. This ensures a stable and secure environment for the study, reducing the risk of disruptions due to software issues.[28]

- Ubuntu has extensive compatibility with a wide range of hardware and software. It supports the latest versions of Python and GNU Radio, which are essential for this study. Additionally, the large user community and extensive documentation provide valuable support and resources for troubleshooting and optimizing the system [28].
- Despite being a powerful and flexible operating system, Ubuntu maintains an intuitive and user-friendly interface. This makes it accessible for users with varying levels of technical expertise, facilitating smoother operation and collaboration within the research team [29].

By leveraging Ubuntu's robust features and advantages, the research team can ensure a stable, secure, and efficient environment for conducting their study.

Etcher

BalenaEtcher, also known as (Etcher), is a free and open-source utility used for burning image files, such as ISO and IMG files, to create bootable USB drives and SD cards. It is developed by Balena, Etcher is a simple and user-friendly tool, making the process of creating bootable media straightforward even for users with limited technical experience.[30]

3.3.5 USRP E312

The USRP E312 is a software-defined radio platform developed by Ettus Research, designed for applications that require a portable and rugged SDR solution. It is part of the USRP family, known for their versatility and wide range of applications in wireless communication research, development, and deployment.

The E312 is designed for standalone operation with an integrated ARM-based processor, allowing users to run custom applications directly on the device without the need for an external computer. It includes a Xilinx Zynq-7020 SoC (System on Chip) that combines a dual-core ARM Cortex-A9 processor with FPGA fabric. This provides both the processing power and flexibility for real-time signal processing.

The E312 supports a frequency range of 70 MHz to 6 GHz, making it suitable for a wide range of wireless communication standards and applications. It offers up to 56 MHz of instantaneous bandwidth. It features multiple connectivity options including USB, Ethernet, and GPIO, allowing for flexible integration into various systems and setups. The E312 is compatible with GNU Radio, a popular open-source toolkit for SDR development making it a perfect tool for this study. [31][32]

Figure 3.4: USRP E312

UHD

The USRP Hardware Driver (UHD) is a comprehensive software framework developed by Ettus Research for managing and controlling the Universal Software Radio Peripheral (USRP) family of software-defined radios (SDRs). UHD provides a standardized interface for users to interact with USRP devices, facilitating tasks such as signal processing, data streaming, and device configuration. UHD abstracts the hardware details of USRP devices, offering a uniform API across different USRP models. This allows users to develop applications that can work with various USRP devices without needing to modify the underlying code for each specific model [33].

UHD is compatible with multiple operating systems, including Linux, macOS, and Windows, ensuring broad accessibility and ease of integration into diverse development environments. It supports high-throughput data streaming between the host computer and the USRP hardware, enabling real-time signal processing applications. It leverages efficient data transfer mechanisms such as direct memory access (DMA) and zero-copy buffers to maximize performance. UHD comes with comprehensive documentation and numerous example programs, aiding users in understanding and utilizing the driver effectively. These resources help both novice and experienced users to quickly get started and develop advanced applications [34].

VERT400 Antenna

The VERT400 antenna is a highly versatile and efficient component designed for the USRP (Universal Software Radio Peripheral), covering 144 MHz, 400 MHz, and 1200 MHz. This tri-band omni-directional vertical antenna is ideal for a wide range of wireless communication applications. [35]

Supporting the 2-meter, 70-centimeter, and 1200-megahertz bands, the VERT400 allows users to operate on different frequencies without switching antennas. Its extended receive range includes 118-160 MHz, 250-290 MHz, 360-390 MHz, 420-470 MHz, 820-960 MHz, and 1260-1300 MHz, ensuring it can capture signals from various sources for enhanced utility in diverse scenarios. Performance-wise, the VERT400 offers a gain of 0 dBi for both the 146 MHz and 446 MHz bands as a 1⁄4 wave antenna, and 3.4 dBi for the 1200 MHz

band with a 5⁄8 wave design. The antenna handles a maximum power of 10 watts, which is suitable for most handheld and portable transceivers. [35]

Including the VERT400 antenna in my study is essential due to its versatility and performance capabilities. Moreover, the compact design and reliable performance make it a practical and effective choice for both experimental and real-world applications. [35]

Figure 3.5: VERT400 Antenna [35]

3.3.6 UAVs

Count	Drone Name	Max Speed
4	CoDrone EDU	2m/s
2	DJI Mini SE	13m/s
1	DJI Mini 2	16m/s

Table 3.1: Overview of drones used for the study

Dji Mini SE

The DJI Mini SE is an entry-level, lightweight drone that offers impressive aerial photography and videography features, making it accessible to beginners and casual users. It combines portability with ease of use, providing a high-quality flying experience. [36]

The DJI Mini SE is an affordable and lightweight drone, weighing just 249 grams, which is perfect for those new to drone flying or looking for a budget-friendly option. It comes equipped with a 12 MP camera and a 1/2.3” CMOS sensor, capable of recording video in 2.7K at 30fps. The drone has a maximum flight time of 30 minutes and can reach speeds of up to 13 m/s in Sport mode. With a transmission range of up to 4 km via enhanced Wi-Fi, the DJI Mini SE can ascend to 3,000 meters above sea level. [36]

Dji Mini 2

The DJI Mini 2 is a compact, lightweight drone designed for both recreational and professional use, offering high-quality aerial photography and videography capabilities. Known for its ease of use and portability, the DJI Mini 2 is an excellent choice for beginners and enthusiasts alike. [37]

The DJI Mini 2 is a highly portable and user-friendly drone, weighing just 249 grams, making it ideal for aerial photography and videography. It features a 12 MP camera with a 1/2.3” CMOS sensor, capable of shooting 4K video at 30fps. The drone offers a maximum

flight time of 31 minutes and a top speed of 16 m/s in Sport mode. It supports a transmission range of up to 10 km through OcuSync 2.0 and can fly up to 4,000 meters above sea level. The DJI Mini 2 also boasts Level 5 wind resistance, ensuring stable flights in windy conditions. [37]

CoDrone EDU

According to [38], CoDrone Edu is designed to facilitate learning in programming and robotics through hands-on projects and exercises. It is specifically tailored for educational purposes, providing an accessible way for students to engage with coding, robotics, and STEM concepts through hands-on learning. CoDrone Edu can be programmed using various languages such as Python, Blockly, and Arduino, making it suitable for different levels of programming expertise. Additionally, it includes sensors and modules for more advanced projects and experiments.

The CoDrone Edu from Robolink is a compact educational drone measuring 130 mm x 130 mm x 45 mm and weighing 46 grams. It offers a flight time of approximately 8 minutes per charge and requires about 40 minutes to recharge its 3.7V 300mAh Li-Po battery. Equipped with a 480p VGA camera, the drone features multiple sensors including a gyroscope, accelerometer, optical flow sensor, and barometer. It connects via Bluetooth 4.0 and supports programming in Python, Blockly, and Arduino, making it versatile for various educational levels. The CoDrone Edu can be controlled using the CoDrone app on Android and iOS devices, with a range of about 50 meters. [38]

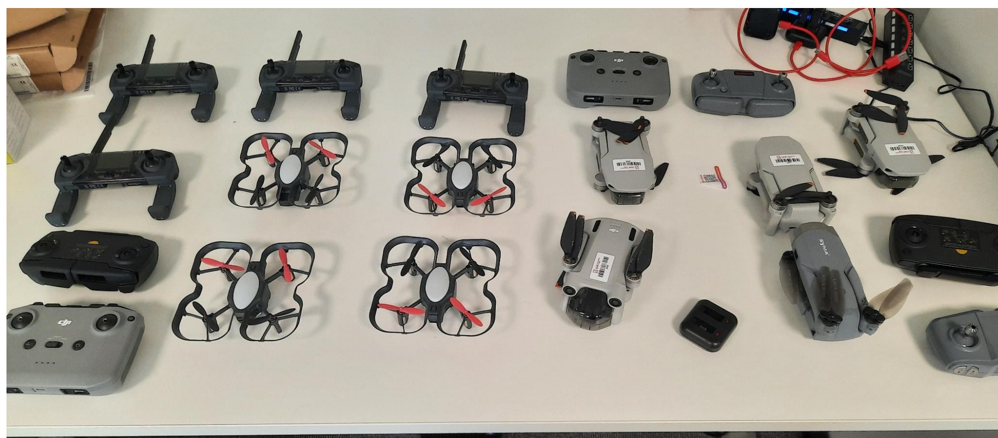


Figure 3.6: Drones used for the study

Chapter 4

Setup & Testing

This chapter provides a comprehensive guide for the installation, configuration, testing, and validation of the hardware and software components required for the study. The successful deployment and operation of the system hinge on the precise execution of these steps to ensure reliability and accuracy in data collection and analysis.

The first section, "Installation and Configuration Guide," details the step-by-step procedures for setting up the necessary software and hardware components. The second section, "Testing and Verification," focuses on validating the setup to ensure all components operate as intended.

4.1 Installation and Configuration Guide

4.1.1 Installation of Ubuntu 24.02

Ubuntu has a step-by-step guide as to how to install Ubuntu on the laptop.[39] I reccomend to check the documentation for a through installation process. First step is to ensure that the laptop is compatible. Website [40] provides with a list of certified hardware which supports Ubuntu. Lenovo Thinkpad T460 (see 3.3.1) has the required capabilities necessary to ensuring reliability. Here are the following step for installing Ubuntu:

- **Download the Ubuntu ISO file:** Download the Ubuntu ISO file from the official Ubuntu website. For this study, I have chosen latest release of Ubuntu 24.04 LTS.
- **Create a bootable USB drive:** Once the download is complete, a bootable USB drive needs to be created. Insert the USB flash drive into the computer and open the etcher. In etcher, select the ISO file, then choose USB drive, and start flash.
- **Boot from the USB drive:** With the bootable USB drive ready, insert it into the computer. Restart the computer and enter the boot menu. Select the USB drive from the list of boot options, which will load the Ubuntu welcome screen.

- **Install Ubuntu:** A prompt appears to choose between Interactive and Automated installation, choose Interactive installation. Interactive is standard, while Automated allows advanced users to import a configuration file for multiple installs. Next step, choose Extended selections; Default includes essentials, and Extended adds additional tools. On the next step, it's recommended to install third-party software for improved device support and additional media formats.
- **Installation Type:** On this step, three options were given - Erase disk and install Ubuntu, Installing Ubuntu alongside another operating system, Manual Installation. Manual Installation was selected as Bitlocker was enabled for my computer. I did not have access to windows operating system to deactivate Bitlocker. Option 1, Erase disk and install Ubuntu, presented issue with bitlocker, as such, manual installation was selected.
- **User Creation:** Follow the on-screen instructions to set the timezone and create an account with username and password.
- **Review and Install:** Final step is to review the settings. If unsatisfied with the settings it is possible to change before installing ubuntu. Once the installation is complete, restart the computer. Remove the USB drive when instructed, and the computer will reboot into Ubuntu.

Upon rebooting, Ubuntu login screen is presented. Log in using the account created during the installation process. To ensure the system is up-to-date, open a terminal and run the following commands:

```
sudo apt update
sudo apt upgrade
```

Listing 4.1: Update and Upgrade Packages

These commands will update the system's package lists and upgrade any installed packages to their latest versions. This is a crucial step as often packages are not at their most recent version. This step keeps the system secure, stable, and up-to-date with the latest features and bug fixes.

4.1.2 Installation of Core Packages (Optional)

This step is optional in the installation process. I previously encountered error with both GNU radio and UHD, hence to ensure error free and stability it was necessary to install dependencies. Most distributions install these packages through package manager when installing GNU radio and UHD (see 4.1.3).

The command `sudo apt install` followed by a long list of packages is used to install all the necessary dependencies and tools for building and running GNU Radio and UHD on

Ubuntu. This command includes development tools (e.g., autoconf, automake, cmake), libraries (e.g., libboost-all-dev, libusb-1.0-0), Python packages (e.g., python3-numpy, python3-requests), and various other required components.

```
sudo apt install autoconf automake build-essential ccache cmake ...
   cpufrequtils ethtool g++ git inetutils-tools libboost-all-dev ...
   libncurses5 libncurses5-dev libusb-1.0-0 libusb-1.0-0-dev libusb-dev...
    python3-dev python3-mako python3-numpy python3-requests python3-...
   scipy python3-setuptools python3-ruamel.yaml git cmake g++ libboost-...
   all-dev libgmp-dev swig python3-numpy python3-mako python3-sphinx ...
   python3-lxml libfftw3-dev libsdl1.2-dev libgsl-dev libqwt-qt5-dev ...
   libqt5opengl5-dev python3-pyqt5 liblog4cpp5-dev libzmq3-dev python3-...
   yaml python3-click python3-click-plugins python3-zmq python3-scipy ...
   python3-gi python3-gi-cairo gir1.2-gtk-3.0 libcodec2-dev libgsm1-dev...
    libusb-1.0-0 libusb-1.0-0-dev libudev-dev python3-setuptools ...
   pybind11-dev python3-matplotlib libsndfile1-dev libsoapysdr-dev ...
   soapysdr-tools python3-pygccxml python3-pyqtgraph libiio-dev ...
   libad9361-dev libspdlog-dev python3-packaging python3-jsonschema ...
   python3-qtpy
```

Listing 4.2: Installing Core packages

4.1.3 Installation of GnuRadio 3.11 and UHD 4.6

After installing To install GNU Radio on modern Ubuntu versions, it's recommended to use pre-built binary packages, which are sufficient for most users. This approach avoids the need to build from source unless the have specific requirements or modifications. Use the following commands:

```
sudo add-apt-repository ppa:gnuradio/gnuradio-releases ppa:...
   ettusresearch/uhd
sudo apt-get update
sudo apt-get install gnuradio libuhd-dev uhd-host python3-uhd uhd-doc ...
   gnuradio-doc gnuradio-dev
```

Listing 4.3: Installing GnuRadio and UHD

4.1.4 Configuration USRP E312

The USRP E312 comes pre-assembled, requiring no additional setup of the motherboard or daughterboard. This makes the initial setup process straightforward and convenient, allowing to focus on deployment and usage rather than assembly.

To begin, connect the power supply and network cables to the USRP E312. The USRP is connect to network using Ethernet cable. Ensure all connections are secure to avoid any interruptions during operation. It's also essential to review and configure any necessary security settings to protect the device and network.

Updating the File System

- **Download the Latest Image:** Install the required version of UHD (v4.6.0) on a host system with Internet access. Execute the following command to download the appropriate image for the E312. This will download the image to `/usr/local/share/uhd/images/usrp_e310_fs.sdimg`.

```
sudo uhd_images_downloader -t sdimg -t e310 -t sg3

```

- **Unmounting SD Card:** It is important to make sure that the SD card is unmounted. First step is to find the location the device is attached to by using the Linux utility `lsblk`. By using `umount` utility, the device is unmounted.

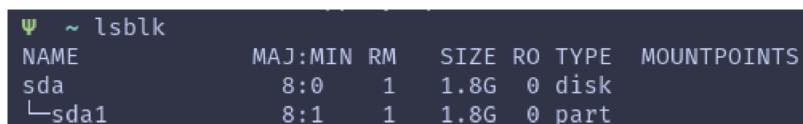


Figure 4.1: Identifying device node attachment point

- **Flash the Image to the SD Card:** Insert the micro SD card into the host system. Use the `dd` command to write the image to the SD card.

```
sudo dd if=/usr/local/share/uhd/images/usrp_e310_fs.sdimg of=/dev/...
    sda bs=1M

```

4.2 Testing and Verification

4.2.1 Secenario 1: Identifying and verifying the operation of USRP E312

USRP devices are identified and addressed using key/value string pairs, which help narrow down the search for specific devices or groups of devices. Most UHD utility applications and examples include an `--args` parameter that accepts a device address. Each USRP device can be identified on the host system using various identifiers.

Identifier	Key	Example Value
Serial Number	serial	135782412
IP Address	addr	128.168.0.221

Table 4.1: Common identifiers and their corresponding keys for USRP

Discovery of devices attached to the system using the `uhd_find_devices` program. This program scans the system for supported devices and outputs a list of discovered devices along with their addresses. The following commands (4.4) help identify and locate the USRP devices connected to the system.

```
uhd_find_devices
uhd_find_devices --args="addr=128.168.0.221"
uhd_find_devices --args="serial=135782412"
```

Listing 4.4: Identifying USRP

Verifying the operation of the USRP is a critical step that ensures functionality, performance, and reliability. The UHD includes a variety of example programs designed to verify the operation of USRP devices. These examples help ensure that the USRP is functioning correctly and performing as expected. The examples files are located in
`/usr/local/lib/uhd/examples`

The `benchmark_rate` example tests the maximum data rates for transmitting (TX) and receiving (RX) samples, helping to ensure that your system can handle the required throughput.

```
./benchmark_rate --rx_rate 10e6 --tx_rate 10e6
```

Listing 4.5: Benchmark Testing

4.2.2 Secenario 2: Testing Signal Transmission and Reception with USRP Using GNU Radio

The test aims to verify the proper functionality of both the USRP and GNU Radio platforms. This test illustrates a fundamental yet an important signal processing task using GNU

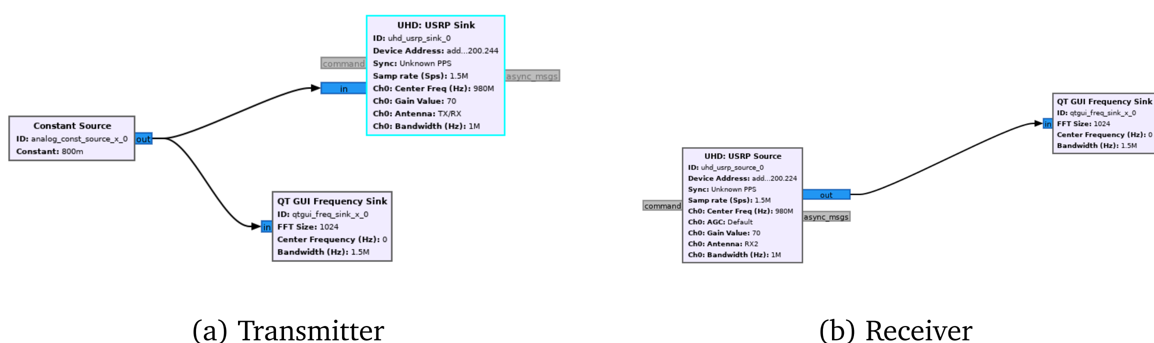
Radio and USRP hardware. By transmitting a constant signal and observing it through frequency spectrum analysis, validates the functionality of the USRP and GNU Radio setup, confirming that both the transmission and reception processes work as intendedW.

The initial phase of the experiment involves setting up the transmission path, as shown in figure 4.2a. The figure illustrates the configuration for transmitting a constant signal. The setup comprises the following components:

- **Constant Source Block:** This block generates a steady signal with a magnitude of 800 millivolts. This unmodulated, continuous signal serves as the input to the transmission chain.
- **USRP Sink Block:** The primary component responsible for emitting the signal into the airwaves. The block takes crucial parameters such as device address, sample rate, Center Frequency, and Gain. For this test, 1.5 million samples per second were configured at 980MHz center frequency with a gain of 70.
- **QT GUI Frequency Sink:** A graphical user interface element that displays the frequency domain representation of the transmitted signal.

The subsequent phase involves setting up the reception path, as depicted in Figure 4.2b. The figure illustrates the configuration for receiving and analyzing the signal. The setup includes the following components:

- **USRP Source Block:** This block is configured to receive signals with same parameters as the USRP sink block.
- **QT GUI Frequency Sink:** Similar to the transmission setup, this graphical interface element visualizes the received signal's frequency spectrum.



(a) Transmitter (b) Receiver

Figure 4.2: Testing Transmission and Reception using USRP

The transmitted signal's frequency spectrum is captured and visualized using the QT GUI Frequency Sink, as shown in Figure 4.3. The figure presents the frequency domain representation of the transmitted signal. The plot depicts the relative gain (dB) as a function of frequency (MHz). The prominent peak at the center frequency (0 MHz in relative terms) corresponds to the constant signal transmitted at 980 MHz. The baseline noise level observed across the spectrum, which is intrinsic to the system and influenced by external

interference. The distinctiveness of the peak indicates a constant signal with minimal frequency deviation, aligning with the expected output from a constant source.

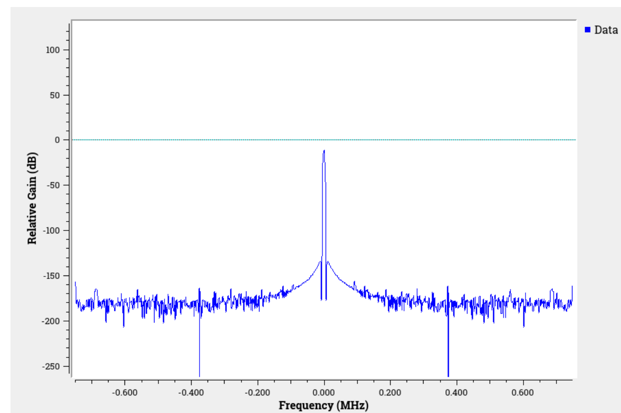


Figure 4.3: Visualisation of received signal

This test confirms the proper operation of both the USRP hardware and GNU Radio software, ensuring their reliability for further research and development. The test lays the groundwork for subsequent development of 5G system that delve into more intricate and sophisticated signal processing techniques.

Chapter 5

System Implementation

5.1 5G System Design

The 5G system design forms the backbone of our wireless sensing application. Central to this design are the transmitter and receiver, implemented using GNU Radio (see 3.3.3). This section describes the configuration and functionality of both the transmitter and receiver, which are critical for the successful implementation of our 5G-based wireless sensing system.

5.1.1 OFDM Transmitter

The GNU Radio flowgraph for the Orthogonal Frequency-Division Multiplexing (OFDM) transmitter is a pivotal element in the dataset collection process aimed at estimating drone counts using 5G Channel State Information (CSI). This flowgraph 5.1 integrates various signal processing blocks to generate and transmit OFDM signals, which are robust against multipath fading and interference. This section provides an in-depth analysis of the transmitter flowgraph, examining its components, configurations, and operational workflow to elucidate its critical role in the experimental setup.

Components and Their Functions

The GNU Radio flowgraph for the OFDM transmitter comprises several key blocks, each serving specific functions to ensure accurate and efficient signal generation and transmission. The primary components include the Constant Source, Stream to Tagged Stream, UChar to Float, OFDM Transmitter, Multiply Const, UHD: USRP Sink, and QT GUI blocks. Each block's parameters and configurations are meticulously set to achieve the desired performance.

The Constant Source Block, identified as `analog_const_source_x_0`, generates a continuous stream of constant data values. This constant data simulates a steady input for the

transmitter, essential for maintaining a reliable data flow. The parameter for this block is set to a constant value of 50, representing a simple data packet or control information.

The Stream to Tagged Stream Block, labeled `blocks_st_tgged_stream_0`, converts this continuous data stream into tagged packets. These tagged packets are crucial for managing packet lengths and synchronization during transmission. The packet length is set to 50, and the length tag key is identified as `packet_len`.

The UChar to Float Block (`blocks_uchar_to_float_0`) converts data from unsigned char to float format, preparing it for subsequent modulation processes. This conversion is necessary because the OFDM modulation requires floating-point arithmetic for accurate signal processing.

The OFDM Transmitter Block (`digital_ofdm_tx_0`) is the core component responsible for generating OFDM signals. This block manages data packetization, modulation, the Inverse Fast Fourier Transform (IFFT), and cyclic prefix addition. Key parameters include an FFT Length of 64, which determines the number of subcarriers (N). The FFT operation can be represented mathematically as:

$$X[k] = \sum_{n=0}^{N-1} x[n] e^{-j2\pi kn/N}$$

where $x[n]$ are the time-domain samples and $X[k]$ are the frequency-domain symbols. The Cyclic Prefix Length is set to 16, which helps mitigate inter-symbol interference (ISI) by adding a prefix that is a copy of the end of the OFDM symbol. This process can be described as:

$$x_{\text{cp}}[n] = x[n + N - L_{\text{cp}}]$$

where L_{cp} is the length of the cyclic prefix. The block also defines Occupied Carriers and Pilot Carriers, specifying which subcarriers are used for data transmission and channel estimation, respectively. Occupied carriers are set as $(-4, -3, -2, -1, 1, 2, 3, 4)$, , and pilot carriers are set as $(-6, -5, 5, 6)$ with pilot symbols as $(-1, 1, -1, 1)$. The modulation schemes for header and payload data are BPSK (Binary Phase-Shift Keying) and QPSK (Quadrature Phase-Shift Keying), respectively, ensuring robust and efficient data encoding.

The Multiply Const Block (`blocks_multiply_const_vxx_0`) scales the amplitude of the OFDM signal, ensuring it is within the desired range for transmission. The multiplication factor is set to 50m, meaning the signal amplitude is scaled by a factor of 0.05.

The UHD: USRP Sink Block (`uhd_usrp_sink_0`) transmits the processed OFDM signal using USRP hardware. It is configured with a device address (addr=128.39.200.106), a sample rate of 1.5M samples per second, a center frequency of 980 MHz, a transmission

gain of 50 dB, and a bandwidth of 1 MHz. These parameters ensure the signal is transmitted at the correct frequency and power level.

Real-time visualization of the transmitted signal is facilitated by the QT GUI blocks, including the QT GUI Time Sink (`qtgui_time_sink_x_0_0_1`), QT GUI Waterfall Sink (`qtgui_waterfall_sink_x_0`), and QT GUI Frequency Sink (`qtgui_freq_sink_x_0_0`). These blocks provide real-time visualizations in the time domain, frequency domain, and spectral view, respectively. The QT GUI Time Sink displays the signal with a sample rate of 1.5M and 1.024k points. The QT GUI Waterfall Sink has an FFT size of 1024, center frequency of 980 MHz, and bandwidth of 750 kHz, while the QT GUI Frequency Sink shares the same center frequency and bandwidth settings.

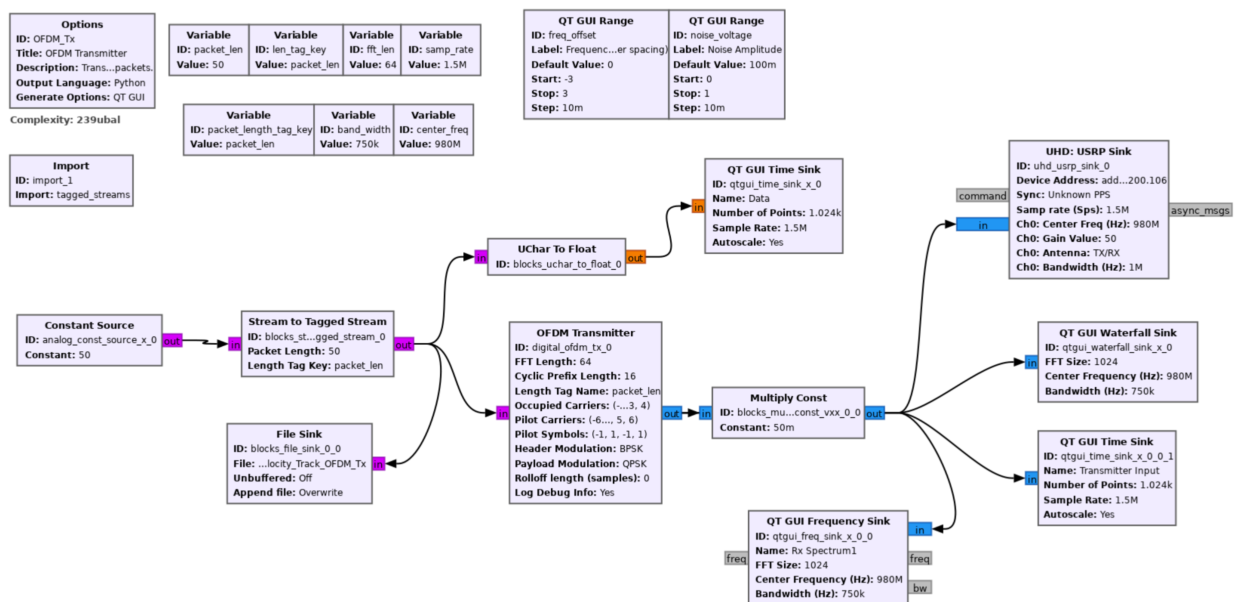


Figure 5.1: OFDM Transmitter Flowchart

Operational Workflow

The operational workflow of the GNU Radio flowgraph for the OFDM transmitter can be broken down into several stages, each responsible for specific signal processing tasks. Initially, the Constant Source Block generates a steady stream of constant data values, serving as the input for the transmitter. This data stream is then converted into tagged packets by the Stream to Tagged Stream Block, ensuring proper management and synchronization of data packets during transmission.

The UChar to Float Block transforms the data packets from unsigned char to float format, preparing them for modulation. Subsequently, the OFDM Transmitter Block processes the float data packets through several critical steps. First, data packets are mapped to modulation symbols (BPSK for headers and QPSK for payloads). The frequency-domain symbols are then converted to time-domain samples using the IFFT process, creating the OFDM signal. A cyclic prefix is added to each OFDM symbol to combat inter-symbol interference.

The Multiply Const Block scales the amplitude of the OFDM signal, ensuring it is appropriate for transmission. The UHD: USRP Sink Block then sends the scaled OFDM signal to the USRP hardware for over-the-air transmission. The USRP is configured with the appropriate sample rate, center frequency, gain, and bandwidth to match the experimental setup.

Real-time visualization of the transmitted signal is provided by the QT GUI Time Sink, Waterfall Sink, and Frequency Sink blocks. These visualizations are essential for monitoring the transmission process, ensuring the integrity and quality of the transmitted signals, and allowing for immediate detection and troubleshooting of any anomalies.

The flowgraph leverages these configurations to transmit OFDM signals using a USRP device. The transmitted signals propagate through the environment, influenced by the presence and movement of drones. The receiver captures these signals, and the collected CSI data reflects the impact of the drones on the signal propagation.

5.1.2 OFDM Receiver

The GNU Radio flowgraph for the Orthogonal Frequency-Division Multiplexing (OFDM) receiver is a critical component in the dataset collection process aimed at estimating drone counts using 5G Channel State Information (CSI). This flowgraph incorporates various signal processing blocks to receive, demodulate, and decode OFDM signals, ensuring accurate data recovery despite the challenges posed by the wireless channel. The OFDM receiver comprises two sections - the OFDM part and the CSI part. This section provides a comprehensive analysis of the receiver flowgraph, focusing on its components, configurations, and operational workflow to highlight its essential role in the experimental setup.

OFDM Section

The OFDM section of the GNU Radio flowgraph is fundamental to demodulating and decoding the received OFDM signals. This process begins with the UHD: USRP Source Block, which is responsible for receiving the incoming RF signals and converting them to baseband. The USRP Source Block is meticulously configured with a sample rate of 1.5M samples per second, a center frequency of 980 MHz, a gain of 70 dB, and a bandwidth of 1 MHz. These parameters ensure that the signals are captured with high fidelity, maintaining the integrity of the received data.

Once the signals are converted to baseband, they are processed by the OFDM Receiver Block. The OFDM Receiver Block (`digital_ofdm_rx_0`) is the core component responsible for demodulating and decoding the received OFDM signals. Key parameters include an FFT Length of 64, consistent with the transmitter's configuration, ensuring proper alignment of subcarriers. The cyclic prefix length is set to 16, matching the transmitter's configuration to effectively handle multipath effects. The block also defines occupied carriers $(-4, -3, -2, -1, 1, 2, 3, 4)$ and pilot carriers $(-6, -5, 5, 6)$, ensuring accurate data recovery

and channel estimation. The pilot symbols are configured as $(-1, 1, -1, 1)$. This block performs several critical functions to decode the received OFDM signals accurately. Initially, the cyclic prefix, which was added during transmission to combat inter-symbol interference (ISI), is removed. This step is crucial for mitigating ISI and preparing the signal for FFT processing. The FFT Length is set to 64, aligning with the transmitter's configuration, ensuring proper demultiplexing of the composite signal into its constituent subcarrier signals.

Real-time visualization of the received signals is provided by several QT GUI blocks, including the QT GUI Time Sink (`qtgui_time_sink_x_0`), QT GUI Waterfall Sink (`qtgui_waterfall_sink_x_0`), and QT GUI Frequency Sink (`qtgui_freq_sink_x_0`). These blocks provide time-domain, frequency-domain, and spectral views of the received signals, respectively, facilitating real-time monitoring of the reception process.

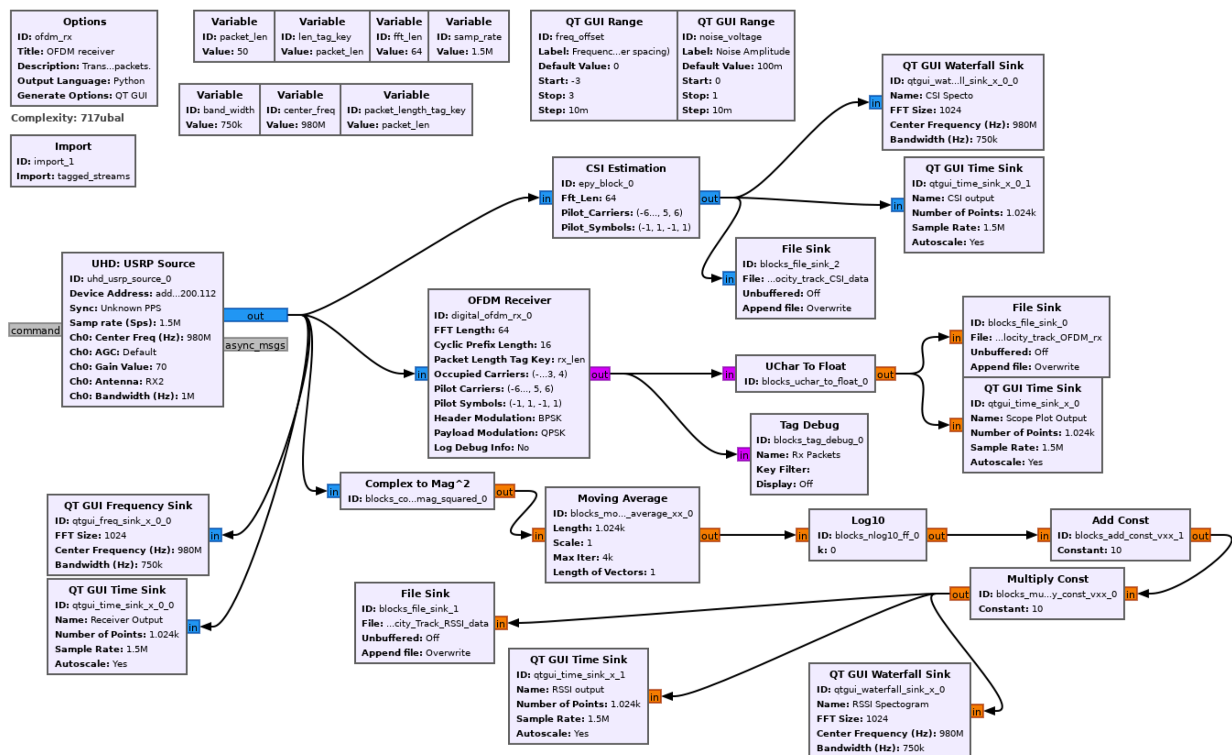


Figure 5.2: OFDM Receiver Flowchart

Operational Workflow

The operational workflow of the GNU Radio flowgraph for the OFDM receiver involves several stages, each responsible for specific signal processing tasks. Initially, the UHD: USRP Source Block captures the incoming RF signals and converts them to baseband. This baseband signal is then fed into the OFDM Receiver Block, where it undergoes several critical processing steps.

First, the cyclic prefix is removed from the received OFDM symbols, mitigating inter-symbol interference. The time-domain symbols are then converted to the frequency domain using the FFT operation:

$$X[k] = \sum_{n=0}^{N-1} x[n] e^{-j2\pi kn/N}$$

where $x[n]$ are the time-domain samples and $X[k]$ are the frequency-domain symbols. The pilot carriers and symbols are used for channel estimation, allowing the receiver to correct for any distortions introduced by the channel.

The demodulated data is then converted from unsigned char to float format by the UChar to Float Block, preparing it for further processing. Real-time visualization of the received signals is provided by the QT GUI Time Sink, Waterfall Sink, and Frequency Sink blocks. These visualizations are crucial for monitoring the reception process, ensuring the integrity and quality of the received signals, and allowing for immediate detection and troubleshooting of any anomalies.

Channel State Information (CSI)

The CSI section of the flowgraph is dedicated to extracting Channel State Information from the received OFDM signals. This information is crucial for understanding how the environment, particularly the presence and movement of drones, affects the signal propagation. The CSI Estimation Block processes the output of the OFDM Receiver Block to estimate the channel characteristics. Configured with an FFT Length of 64, the block uses the same pilot carriers and symbols as the OFDM Receiver Block, ensuring consistent and accurate channel estimation.

Channel estimation involves comparing the received pilot subcarriers with the known pilot symbols, allowing the block to determine the channel's effect on the transmitted signal. This process is critical for analyzing the impact of drone movements on the transmitted signals. The extracted CSI data provides a detailed representation of the channel's behavior, which is essential for further analysis and model training in the research. The CSI data enables the researchers to understand the multipath effects, signal fading, and other channel impairments introduced by the drones, facilitating the development of robust algorithms for drone detection and estimation.

Detailed Analysis of CSI Estimation The process begins with the CSI Estimation Block receiving input symbols, which are complex values representing the received OFDM symbols. The input symbols are divided into chunks corresponding to the FFT length (64 in this case). These symbols are then reshaped into a matrix where each row represents an OFDM symbol, and each column corresponds to a subcarrier.

$$\text{in0} = \text{input_items}[0]$$
$$\text{num_symbols} = \frac{\text{len(in0)}}{\text{fft_len}}$$

$$\text{in0} = \text{in0}[:\text{num_symbols} \times \text{fft_len}].\text{reshape}((\text{num_symbols}, \text{fft_len}))$$

The pilot carriers are strategically placed within the subcarriers. For example, if the pilot carriers are $[-6, -5, 5, 6]$, they are offset by half the FFT length to map them into the correct positions within the FFT output:

$$\text{pilot_indices} = \text{pilot_carriers} + \frac{\text{fft_len}}{2}$$

$$\text{pilot_indices} = [-6 + 32, -5 + 32, 5 + 32, 6 + 32] = [26, 27, 37, 38]$$

These indices must be within the valid range of [0, FFT Length), ensuring no out-of-bound errors. The corresponding pilot symbols are $[-1, 1, -1, 1]$.

Next, the block extracts the received pilot symbols from the incoming OFDM symbols at these pilot positions. The channel response at the pilot subcarriers is estimated by dividing the received pilot symbols by the known transmitted pilot symbols:

$$H_{\text{estimated}}[i, \text{pilot_indices}] = \frac{\text{received_pilots}}{\text{pilot_symbols}}$$

This estimation is performed for each OFDM symbol, resulting in a channel estimate matrix $H_{\text{estimated}}$ of size $(\text{num_symbols}, \text{fft_len})$. The estimated channel response at the pilot subcarriers is then interpolated across all subcarriers to obtain a full channel estimate for each OFDM symbol. Linear interpolation is used, where the interpolated channel response for subcarrier k is:

$$H_{\text{interpolated}}[i, k] = \text{interp}(\text{indices}, \text{pilot_indices}, H_{\text{estimated}}[i, \text{pilot_indices}])$$

where indices is an array of all subcarrier indices $[0, 1, ..., \text{fft_len} - 1]$.

Finally, the full channel estimate $H_{\text{interpolated}}$ is reshaped back into a flat array to match the output format of the block, ensuring compatibility with downstream processing blocks in the flowgraph.

The CSI Estimation Block's detailed process ensures accurate channel state information extraction from the received OFDM signals. By leveraging pilot subcarriers and known pilot symbols, the block estimates the channel response, crucial for analyzing the impact of environmental factors, such as drone movements, on signal propagation.

Significance and Impact

The GNU Radio flowgraph for the OFDM transmitter and receiver is integral to the dataset collection process for several reasons. First, it ensures the generation and reception of high-quality OFDM signals, which are resilient against multipath fading and interference,

essential for reliable data transmission and recovery. The proper packetization, modulation, and cyclic prefix addition within the transmitter flowgraph guarantee that transmitted data can be accurately received and decoded by the receiver, facilitating precise CSI estimation.

The inclusion of QT GUI blocks for real-time visualization allows researchers to continuously monitor both the transmission and reception processes, ensuring signal integrity and quality. The transmitter flowgraph's flexible design permits easy adjustments to parameters such as FFT length, cyclic prefix length, modulation schemes, and transmission gain, making it adaptable to various experimental conditions. Similarly, the receiver flowgraph allows adjustments to parameters such as FFT length, cyclic prefix length, and gain settings, ensuring adaptability and accuracy in data recovery.

In conclusion, the GNU Radio flowgraph for both the OFDM transmitter and receiver is a sophisticated and essential component of the experimental setup for estimating drone counts using 5G CSI. The detailed configuration and robust processing pipeline ensure the reliable generation, transmission, reception, and decoding of OFDM signals, ultimately contributing to the accuracy and reliability of the collected dataset. This in-depth analysis underscores the importance of each component and the intricate interactions that make the flowgraph an effective tool for advanced wireless communication experiments.

5.2 Dataset Collection

The success of any data-driven research relies on the quality of the data collected. In this section the methodologies and processes employed for collecting and analyzing the data necessary for counting the number of drones is presented. The aim is to ensure that the data is accurate, relevant, and sufficient to support model development. This section also discusses the specific parameters and configurations used during the data collection phase to ensure consistency and reliability.

The dataset, consisting of Channel State Information (CSI) values, is crucial for the accurate training and validation of the model. The data collection process involved through planning and execution to ensure the integrity and reliability of the data.

The core components used for data collection were two USRP E312s (Universal Software Radio Peripheral), one designated as the transmitter and the other as the receiver. These devices were strategically placed 4 meters apart to create a controlled environment for capturing the CSI values accurately. The drones were flown within this setup to simulate real-world scenarios while ensuring that the collected data was relevant for the intended application.

The physical setup involved securing the USRP devices to stable platforms to prevent any movement during the experiments. The transmitter was configured to emit 5G signals, while the receiver was set up to capture the CSI values generated by the interaction of

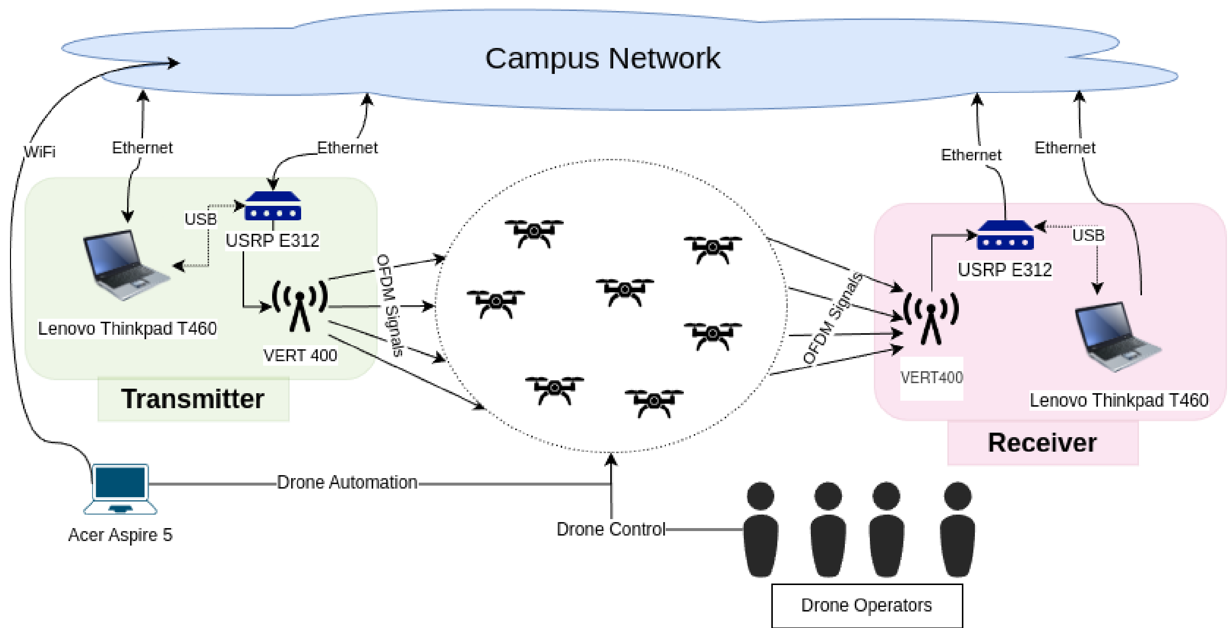


Figure 5.3: Dataset collection system overview

these signals with the flying drones. Ensuring a stable and interference-free environment was paramount to obtaining high-quality data.

The data collection process was designed to systematically gather CSI values as different numbers of drones were flown within the experimental setup. The procedure began with flying a single drone for a duration of 30 seconds. This initial experiment established a baseline for the CSI values with minimal interference. Subsequently, additional drones were introduced one at a time, up to a maximum of seven drones, each flown for the same duration.

A total of seven drones were used in the experiments: four Edu CoDrone drones and three DJI Mini drones. The Edu CoDrone drones were programmed to move at speeds ranging from 0.1 to 2 meters per second in a sway pattern. These drones were placed on the floor facing different directions, causing them to sway in various directions. The DJI Mini drones were manually controlled, with human operators determining their speed and direction during the flights. All drones were flown at different altitudes to simulate diverse real-world conditions.

To simulate real-world conditions, the drones were flown at different speeds and followed varied paths. This was critical to ensure that the collected data reflected the complexities of real-world scenarios. The Edu CoDrone drones, which moved in a sway pattern, provided consistent, programmable movements, while the manually controlled DJI Mini drones introduced variability in speed and direction, as determined by the operators. The speeds of the drones ranged from slow hovering (around 0.1 meters per second) to more rapid movements (up to 2 meters per second). The paths included straight lines, curves, circular patterns, and random movements to simulate different flight behaviors.

The experiments were conducted in an indoor classroom at the university, ensuring controlled environmental conditions. The classroom was cleared of tables and chairs to provide a spacious area for the drones to fly without obstructions. This controlled environment helped minimize external interferences and maintain consistency across all experiments.

5.3 CNN Model Design

In this chapter, we delve into the development of the machine learning model designed to estimate the drone count using wireless sensing data collected via 5G technology. The integration of machine learning, particularly Convolutional Neural Networks (CNNs), is a critical aspect of this research, as it allows for the accurate analysis and interpretation of complex data patterns. This chapter outlines the comprehensive process involved in developing the machine learning model, data preparation and preprocessing, feature engineering, model construction, and training.

We begin by discussing the detailed steps for preparing and preprocessing the data to ensure its integrity, and suitable for training the model. Next, we cover the feature engineering process, where relevant features are extracted and transformed to enhance the model's performance. We then move on to the construction of the CNN model, describing the architecture, layer configuration, and parameters used. The model training process is elaborated upon, including the methods employed for optimizing the model.

5.3.1 Data Loading and Preprocessing

The `DataLoader` class is a crucial component of the preprocessing phase, designed to handle the loading, normalization, and preparation of the Channel State Information (CSI) data for training and evaluation of the neural network model. This section describes the functionality and implementation details of the `DataLoader` class, which facilitates efficient and standardized data preprocessing.

Class Initialization

The initialization of the `DataLoader` class sets up essential parameters and attributes required for data handling. The constructor (`__init__` method) accepts three parameters.

- `base_path`: Specifies the directory where the data files are stored.
- `samples_to_take`: Indicates the number of samples to be extracted from each file for processing.
- `sequence_length`: Defines the length of each sequence to be used as input to the neural network.
- `data_list`: A list to store the loaded and processed data sequences.

- `labels_list`: A list to store the corresponding labels for each data sequence.

```
class DataLoader:
    def __init__(self, base_path, samples_to_take=1_000_000, ...
sequence_length=256):
        self.base_path = base_path
        self.samples_to_take = samples_to_take
        self.sequence_length = sequence_length
        self.data_list = []
        self.labels_list = []
```

Listing 5.1: Initializing Dataloader class

Data Loading Method

The `load_data` method is responsible for reading the CSI data files, normalizing the data, and organizing it into a format suitable for model training. This method processes each file sequentially and extracts the necessary information.

```
def load_data(self):
    for i in range(1, 8):
        file_path = f"{self.base_path}{i}"
        label = i - 1
        file_data = np.fromfile(open(file_path, 'rb'), dtype=np....
complex64)
        start_idx = len(file_data) // 2 - self.samples_to_take // 2
        end_idx = start_idx + self.samples_to_take
        real_part = file_data.real[start_idx:end_idx]
        imag_part = file_data.imag[start_idx:end_idx]
        abs_part = np.abs(file_data)[start_idx:end_idx]
        combined_data = np.column_stack((real_part, imag_part, abs_part...
))
        self.data_list.append(combined_data)
        self.labels_list.extend([label] * len(combined_data))
    self.data = np.vstack(self.data_list)
    self.labels = np.array(self.labels_list)
    self.reshape_data()
```

Listing 5.2: Method for reading CSI file

The process begins with the method that iterates through the data files, which are assumed to be named sequentially. Each file is read as a binary file with complex64 data type using `np.fromfile`. A subset of the data is extracted based on `samples_to_take`, ensuring a central segment of the data is used. The real part, imaginary part, and absolute value of the complex data combined into a single array using `np.column_stack`. Labels corresponding to the number of drones are appended to `labels_list`. The combined data is reshaped to the appropriate format by calling the `reshape_data` method.

Data Normalization

The `normalize` method standardizes the data by adjusting the mean to 0 and the standard deviation to 1. This normalization process is crucial for ensuring that the neural network can effectively learn from the data.

$$\text{Normalized Data} = \frac{X - \mu}{\sigma}$$

where X represents the original data, μ is the mean of the data, and σ is the standard deviation of the data.

Data Reshaping

The `reshape_data` method restructures the data into sequences of a specified length, making it suitable for input into the neural network. This method also ensures that the labels are correctly aligned with the reshaped data sequences.

```
def reshape_data(self):
    num_samples = self.data.shape[0] // self.sequence_length
    self.data = self.data[:num_samples * self.sequence_length]
    self.labels = self.labels[:num_samples * self.sequence_length]
    self.data = self.data.reshape(num_samples, self.sequence_length, 3)
    self.labels = self.labels[::self.sequence_length]
```

The method calculates the number of complete sequences that can be formed from the data. The data is truncated to ensure it fits into the calculated number of sequences. The data is reshaped into the format `(num_samples, sequence_length, 3)`, where 3 represents the three components (real, imaginary, absolute). Labels are aligned with the reshaped data sequences to ensure correct labeling.

Train-Test Split

The `get_train_test_split` method divides the data into training and testing sets, facilitating model evaluation.

```
def get_train_test_split(self, test_size=0.2, random_state=42):
    return train_test_split(self.data, self.labels, test_size=test_size...
    , random_state=random_state)
```

The method uses `train_test_split` from `sklearn.model_selection` to split the data and labels into training and testing sets. The method allows specification of the test set size (`test_size`) and a random seed (`random_state`) for reproducibility.

To summarize, the `DataLoader` class is an essential component for preprocessing CSI data, ensuring that the data is properly normalized, reshaped, and split for effective training

and evaluation of the neural network model. By automating these preprocessing steps, the `DataLoader` class enhances the reproducibility and reliability of the data pipeline, providing a robust foundation for subsequent model training and evaluation phases.

5.3.2 CNN Model Architecture

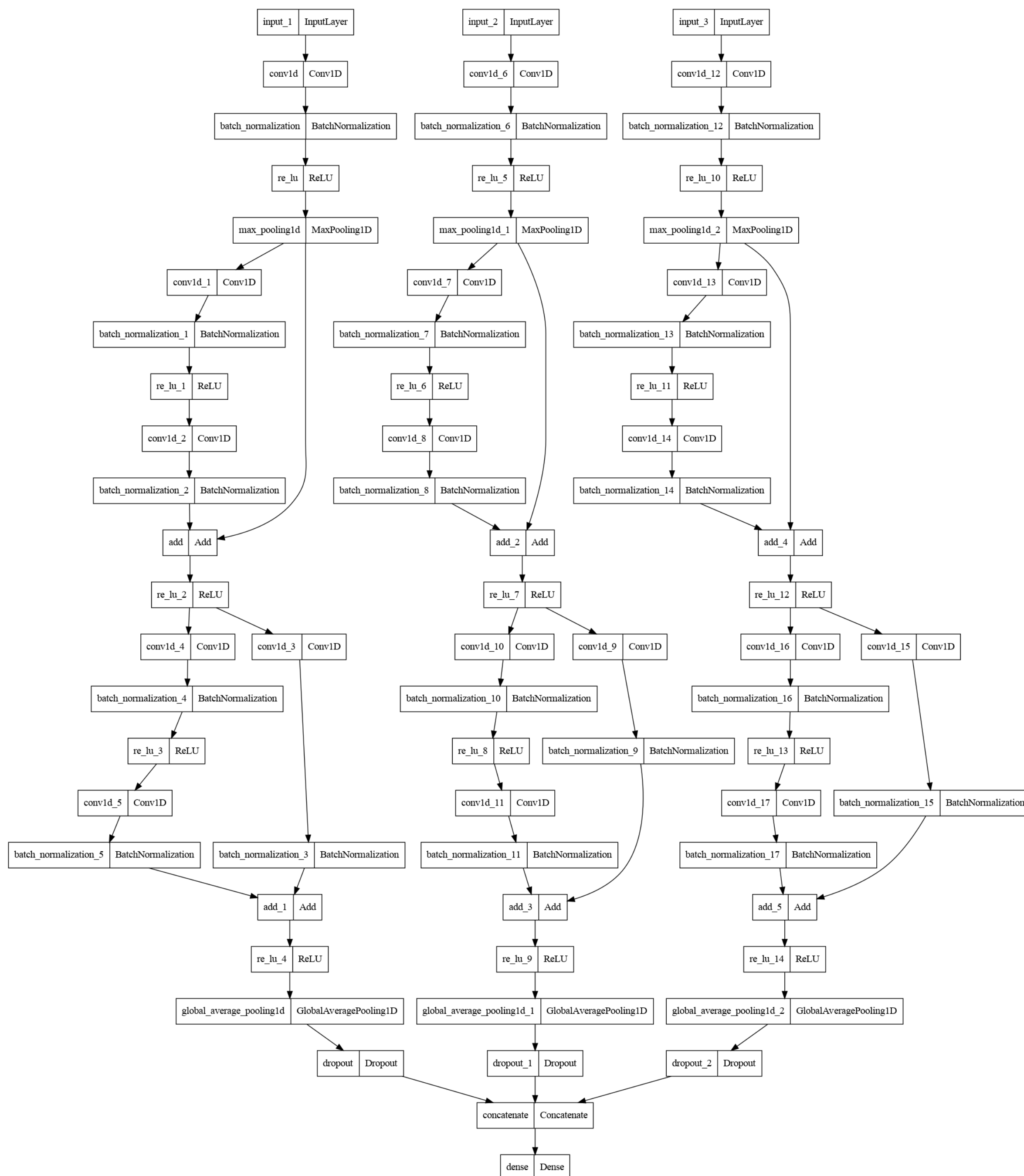


Figure 5.4: CNN model architecture

The proposed model is a Residual Network (ResNet) designed for classifying the number of drones based on the Channel State Information (CSI) data represented as complex numbers. This model leverages the powerful feature extraction capabilities of convolutional neural networks (CNNs) and the efficient gradient propagation of residual networks. The model processes three distinct input features—real part, imaginary part, and absolute

value of the complex numbers—through parallel input branches. Each branch applies a series of convolutional and residual blocks to extract high-level features before these features are concatenated and classified by a fully connected layer.

The CSI data for each drone is preprocessed into three separate components: the real part, the imaginary part, and the absolute value of the complex numbers. These components are structured into sequences of fixed length, serving as inputs to the neural network. The input shape for each component is defined as $(sequence_length, 1)$, where the sequence length is a hyperparameter determined based on the characteristics of the data.

```
inputs = layers.Input(shape=input_shape)
```

The network architecture consists of three parallel branches, each dedicated to one of the input components. The structure of each branch is described below:

Initial Convolutional Layer: Each branch begins with a one-dimensional convolutional layer (Conv1D) with 8 filters, a kernel size of 7, and a stride of 2. This layer is followed by batch normalization and a ReLU activation function. The convolutional layer aims to capture local temporal patterns in the input sequences. The output of this layer is then downsampled using a max-pooling layer with a pool size of 3 and a stride of 2 to reduce the computational complexity and the risk of overfitting.

Residual Blocks: Residual blocks form the core of the network, allowing it to learn deeper representations while mitigating the vanishing gradient problem. Each residual block comprises:

- A Conv1D layer with batch normalization and ReLU activation.
- A second Conv1D layer with batch normalization.
- A shortcut connection that adds the input of the block to the output of the second Conv1D layer. This shortcut connection may include a Conv1D layer with a kernel size of 1 if the input and output dimensions differ.
- The output of the residual block is obtained by adding the shortcut connection to the output of the second Conv1D layer, followed by a ReLU activation.

The number of filters in the Conv1D layers increases as the network deepens, starting from 8 and doubling at each downsampling stage. The use of residual connections ensures that the gradient can flow back through the network effectively, facilitating the training of deeper models.

Global Average Pooling and Dropout: After passing through the residual blocks, the output is subjected to global average pooling, which reduces each feature map to a single

value. This operation preserves spatial information while significantly reducing the number of parameters. To further prevent overfitting, a dropout layer with a dropout rate of 0.5 is applied.

Feature Concatenation and Classification: The outputs of the three parallel branches are concatenated, combining the extracted features from the real part, imaginary part, and absolute value of the input sequences. The concatenated features are then fed into a fully connected dense layer with a softmax activation function, which outputs the probability distribution over the seven classes (number of drones).

```
def residual_block(self, x, filters, kernel_size=3, stride=1, ...
    conv_shortcut=False):
    shortcut = x
    if conv_shortcut:
        shortcut = layers.Conv1D(filters, 1, strides=stride, ...
    kernel_regularizer=regularizers.l2(0.001))(shortcut)
        shortcut = layers.BatchNormalization()(shortcut)
    x = layers.Conv1D(filters, kernel_size, strides=stride, padding='...
    same', kernel_regularizer=regularizers.l2(0.001))(x)
    x = layers.BatchNormalization()(x)
    x = layers.ReLU()(x)
    x = layers.Conv1D(filters, kernel_size, padding='same', ...
    kernel_regularizer=regularizers.l2(0.001))(x)
    x = layers.BatchNormalization()(x)
    x = layers.add([shortcut, x])
    x = layers.ReLU()(x)
    return x

def build_model(self):
    input_layers = []
    outputs = []

    for _ in range(self.channels):
        inputs = layers.Input(shape=self.input_shape)
        x = layers.Conv1D(8, 7, strides=2, padding='same', ...
    kernel_regularizer=regularizers.l2(0.001))(inputs)
        x = layers.BatchNormalization()(x)
        x = layers.ReLU()(x)
        x = layers.MaxPooling1D(3, strides=2, padding='same')(x)

        x = self.residual_block(x, 8)
        x = self.residual_block(x, 16, stride=2, conv_shortcut=True)
        x = self.residual_block(x, 16)
        x = self.residual_block(x, 32, stride=2, conv_shortcut=True)

        x = layers.GlobalAveragePooling1D()(x)
        x = layers.Dropout(0.5)(x)
        input_layers.append(inputs)
```

```
        outputs.append(x)

    merged = layers.concatenate(outputs)
    output = layers.Dense(self.num_classes, activation='softmax')(...
  merged)
    model = models.Model(inputs=input_layers, outputs=output)
    return model
```

Listing 5.3: Resnet Code

The model is compiled using the Adam optimizer with a learning rate of 1×10^{-4}. The loss function used is sparse categorical crossentropy, suitable for multi-class classification problems. The model is trained with early stopping and learning rate reduction on plateau callbacks to prevent overfitting and to ensure efficient convergence. Early stopping monitors the validation loss and halts training if no improvement is observed for 10 consecutive epochs. The learning rate reduction on plateau reduces the learning rate by a factor of 0.5 if the validation loss does not improve for 5 consecutive epochs, with a minimum learning rate of 1×10^{-7}.

```
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy',...
     metrics=['accuracy'])

history = model.fit(X_train, y_train, validation_data=(X_test, y_test),...
     epochs=100, batch_size=64, callbacks=callbacks_list)
```

The model's performance is evaluated on a separate test set, and the best model based on validation loss is saved. Additionally, the training history, including accuracy and loss curves, is plotted to visualize the model's performance over the training epochs.

Chapter 6

Results

6.1 Training and Validation Loss

The training and validation loss curves for the ResNet model are presented in Figure 6.1. The loss curves demonstrate the model's learning process over 75 epochs. Initially, both the training and validation losses decrease sharply, indicating that the model is learning effectively. The training loss continues to decrease steadily, suggesting that the model fits the training data well. However, the validation loss starts to plateau around epoch 30, indicating that the model begins to generalize to the validation data.

Despite slight fluctuations in the validation loss beyond epoch 30, it remains relatively low and close to the training loss, indicating that the model is not significantly overfitting. The final validation loss is approximately 0.05, demonstrating the model's ability to generalize to unseen data.

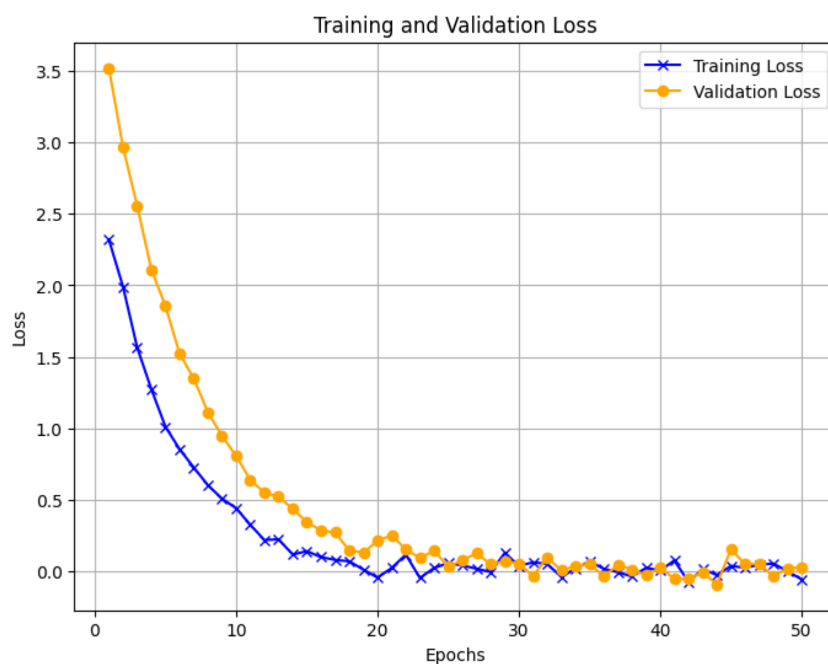


Figure 6.1: Training and Validation Loss

6.2 Training and Validation Accuracy

The training and validation accuracy curves for the ResNet model are shown in Figure 5.2. These curves highlight the model's performance in terms of correctly classifying the number of drones. Similar to the loss curves, the accuracy curves show a rapid increase in both training and validation accuracy during the initial epochs.

The training accuracy reaches to 100%, indicating that the model can almost perfectly classify the training data. The validation accuracy also increases steadily, peaking at approximately 90% around epoch 30. This high validation accuracy demonstrates the model's robust performance and its ability to generalize well to the validation dataset.

The slight decline in both training and validation accuracy beyond epoch 50 suggests that the learning rate may need to be reduced further, or that the model could benefit from additional regularization to prevent overfitting. However, the overall high accuracy indicates the effectiveness of the model in classifying the number of drones based on the CSI data.

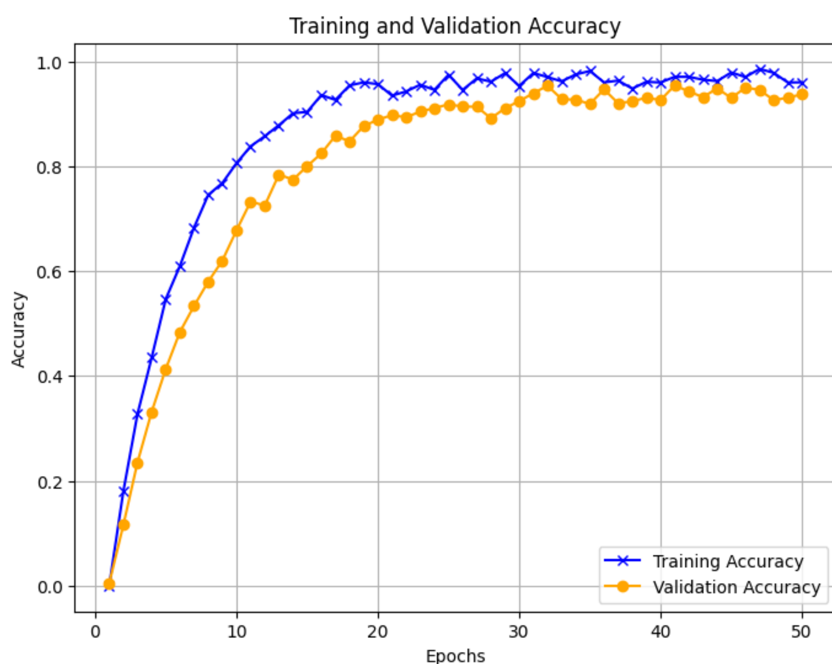


Figure 6.2: Training and Validation Accuracy

6.3 Confusion Matrix

The confusion matrix for the ResNet model is presented in Figure 5.3. The confusion matrix provides a detailed breakdown of the model's classification performance across all classes (i.e., the number of drones). The rows of the matrix represent the actual number of drones, while the columns represent the predicted number of drones.

The diagonal elements of the confusion matrix indicate the number of correctly classified instances for each class, while off-diagonal elements represent misclassifications. The

model demonstrates high classification accuracy across all classes, with most of the confusion matrix's values concentrated along the diagonal.

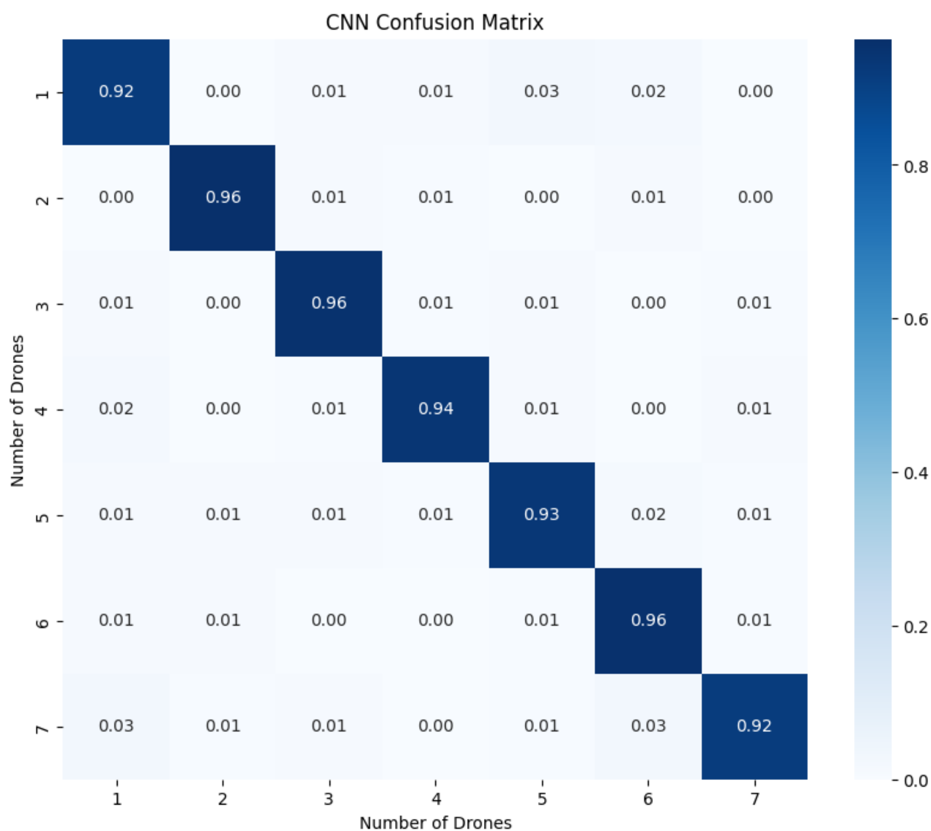


Figure 6.3: CNN Confusion Matrix

Drones	Overall Accuracy
1	92%
2	96%
3	96%
4	94%
5	93%
6	96%
7	92%

Table 6.1: Overall Accuracy of the CNN Model

These results indicate that the model performs well in distinguishing between different numbers of drones, with relatively few misclassifications. However, there are minor confusions observed between adjacent classes, suggesting that the model occasionally struggles with differentiating between similar numbers of drones.

The table 6.2 summarizes the Precision, Recall, and F1 Score for each class (number of drones) based on the confusion matrix. The provided metrics indicate that the CNN model generally performs well across all classes, but there are some variations. The model shows very high precision and recall for classifying two and three drones, indicating that it is highly accurate in both identifying these instances and not misclassifying others as such.

For most classes, the precision, recall, and F1 scores are above 0.9, indicating strong overall performance. The model is reliable in identifying and correctly classifying different numbers of drones. The precision and recall for one and seven drones are slightly lower compared to other classes. This suggests that the model may occasionally misclassify these instances or miss some actual instances.

Drones	Precision	Recall	F1 Score
1	0.929	0.920	0.924
2	0.970	0.960	0.965
3	0.960	0.960	0.960
4	0.949	0.940	0.945
5	0.930	0.930	0.930
6	0.960	0.960	0.960
7	0.911	0.920	0.915

Table 6.2: Precision, Recall, and F1 Score of the CNN model

This implies that the CNN model is robust and performs well in classifying multiple drones, particularly when there are two, three, or six drones, where it achieves the highest F1 scores. The slight drop in precision and recall for one and seven drones suggests room for improvement. Further training or adjustments to the model may help in enhancing the accuracy for these specific cases. With F1 scores consistently above 0.9 for all classes, the CNN model is reliable for practical applications where accurate classification of the number of drones is critical.

Chapter 7

Discussions

7.1 Interpretation of Results

The results demonstrate that integrating 5G CSI with a Multimodal 1D-CNN model significantly enhances the accuracy of drone detection and counting. The high resolution and precision of 5G CSI allow for detailed analysis of the propagation environment, enabling the model to identify subtle patterns indicative of drone presence. The 1D-CNN model effectively learns and extracts features from the CSI data, achieving high accuracy across various experimental scenarios.

The ResNet model trained on the drone CSI data exhibits strong performance, achieving high accuracy on both training and validation datasets. The loss curves and accuracy metrics indicate effective learning and generalization, while the confusion matrix confirms robust classification capabilities with minimal misclassifications across all classes.

Metrics such as accuracy, precision, recall, and F1-score show substantial improvement over traditional methods like radar and visual surveillance. These results align with previous studies highlighting the potential of machine learning techniques in enhancing wireless sensing capabilities. The findings confirm that advanced machine learning models, combined with high-resolution 5G data, can overcome the limitations of conventional drone detection methods.

Further improvements could be achieved by fine-tuning the learning rate schedule, increasing the dataset size, or exploring additional regularization techniques. These strategies could help reduce slight fluctuations observed in the validation loss and enhance the model's robustness.

Overall, the results validate the effectiveness of using a ResNet architecture for time-series classification tasks involving complex CSI data. The model's strong performance provides a solid foundation for future research and potential real-world applications in the field of drone detection.

7.2 Comparison with Existing Studies

Comparing the findings with existing literature, it is evident that the use of 5G CSI provides a significant advantage over other wireless sensing technologies. Previous research has demonstrated the limitations of Wi-Fi and Bluetooth-based sensing in terms of range and accuracy [41, 42]. The enhanced capabilities of 5G, particularly its ability to provide detailed CSI data, contribute to the improved performance observed in this study.

Furthermore, the application of CNNs in wireless sensing is supported by several studies that have shown their effectiveness in pattern recognition tasks [43, 44]. The use of a Multimodal 1D-CNN in this research further validates the model's capability to handle complex time-series data and extract meaningful features for drone detection.

7.3 Challenges

One significant challenge during the experiments was the collision of drones, particularly because some required manual control, increasing the risk of accidents. These collisions posed risks to equipment and introduced noise into the dataset, potentially compromising CSI data quality. To mitigate this, operators received additional training, and safety protocols, including physical barriers, were enhanced, reducing collisions and improving data quality.

Network connectivity issues also posed challenges, with laptops struggling to connect to USRP devices due to closed Ethernet ports. This disrupted data collection and timing. The network configuration was reviewed and adjusted, and backup equipment was kept ready to ensure continuity of the experiments.

The varying battery life of drones was another challenge, causing delays due to frequent battery changes and introducing environmental noise. A rotation system with fully charged spare batteries and regular maintenance checks helped reduce downtime and maintain consistency.

Developing the ResNet model for drone classification using CSI data involved managing and preprocessing complex datasets, balancing model complexity and generalization, and integrating multiple input channels. Techniques like normalization, dropout, and batch normalization were critical but required extensive experimentation to optimize. Despite these challenges, a systematic and iterative approach enabled the creation of an effective model.

Chapter 8

Conclusions

This thesis has explored the integration of 5G Channel State Information (CSI) with advanced machine learning techniques, specifically a Multimodal 1D Convolutional Neural Network (1D-CNN), to develop a robust system for estimating drone counts in various environments. The research was driven by the increasing challenges posed by the proliferation of drones in sectors such as agriculture, surveillance, and delivery services, which require effective monitoring and management to ensure safety, security, and compliance with regulations.

The study began with the collection and preprocessing of high-resolution 5G CSI data, addressing the complexities of managing and normalizing this data to extract meaningful features. By leveraging the powerful pattern recognition capabilities of CNNs, the developed model was able to accurately detect and count drones, demonstrating substantial improvements over traditional methods like radar and visual surveillance.

Key findings highlighted the significant advantages of using 5G technology for wireless sensing applications, with CSI providing detailed insights into the propagation environment that are crucial for accurate drone detection. The 1D-CNN model's ability to handle complex time-series data and extract relevant features further validated the potential of machine learning in enhancing wireless sensing capabilities.

Despite the promising results, the research also identified several limitations, including the controlled experimental setup that may not fully represent real-world complexities, and the computational demands of processing high-dimensional CSI data. Future research should focus on conducting experiments in diverse environments, optimizing models for real-time applications.

In conclusion, this thesis contributes to the advancement of wireless sensing technologies by demonstrating the feasibility and effectiveness of integrating 5G CSI with CNNs for drone detection. The findings provide valuable insights for the development of sophisticated monitoring systems and set a foundation for future innovations in the field.

Chapter 9

Additional Work

9.1 Capturing Received Signal Strength Values

The RSSI section of the flowgraph measures the received signal strength and provides visualizations of the signal power. This part of the flowgraph includes several blocks that compute, average, and visualize the RSSI values. The process begins with the Complex to Mag^2 Block, which calculates the magnitude squared of the complex received signals. This calculation is essential for understanding the strength and quality of the received signals, as it provides a direct measure of signal power. The operation is represented by $|X[k]|^2 = \Re(X[k])^2 + \Im(X[k])^2$, where $\Re(X[k])$ and $\Im(X[k])$ are the real and imaginary parts of the received signals, respectively [45].

The Moving Average Block smooths the data by averaging the signal power over a length of 1.024k, reducing noise and providing a more stable measure of signal strength. This averaging process is crucial for mitigating the effects of short-term fluctuations and providing a clearer representation of the signal's overall strength. The averaged signal power is then converted to a logarithmic scale using the Log10 Block. The logarithmic transformation is represented by $P_{\text{dB}} = 10 \log_{10}(P_{\text{linear}})$, facilitating easier analysis and visualization [46].

To ensure the signal power is within a suitable range for visualization and analysis, the logarithmic power measure is adjusted by adding a constant value of 10 using the Add Const Block and scaling by a factor of 10 using the Multiply Const Block. These adjustments are essential for standardizing the signal power measurements, making them easier to compare and analyze across different conditions [47].

The processed RSSI data is saved to a file using the File Sink Block, enabling further analysis and archiving of the signal strength data. Real-time visualization is provided by the QT GUI Time Sink and Waterfall Sink blocks, which display the RSSI values in the time domain and frequency domain, respectively. The QT GUI Time Sink visualizes the RSSI values with a sample rate of 1.5M and 1.024k points, while the QT GUI Waterfall Sink provides a spectral view with an FFT size of 1024, center frequency of 980 MHz,

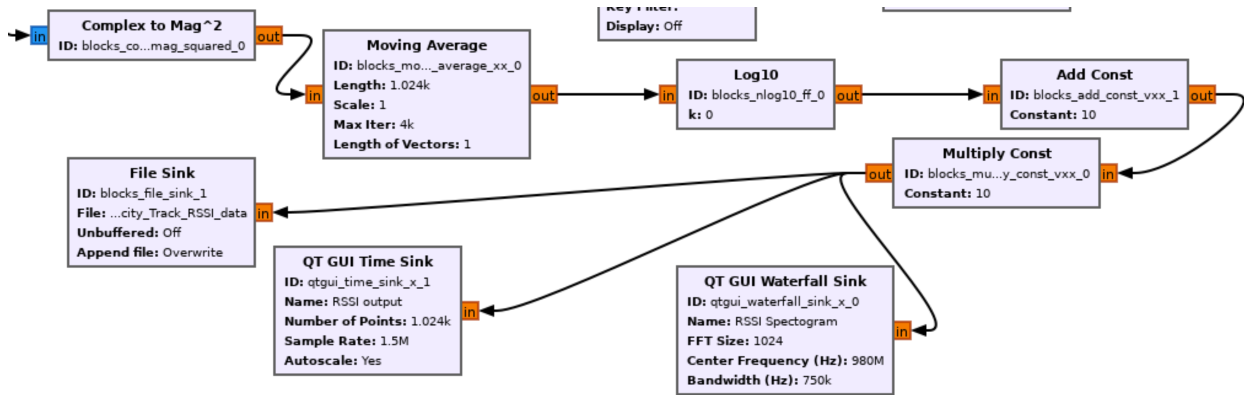


Figure 9.1: Flowchart of RSSI

and bandwidth of 750 kHz. These visualizations are crucial for monitoring the reception process, ensuring signal integrity and quality, and allowing for immediate detection and troubleshooting of any anomalies [48].

Bibliography

[1] Dimosthenis C. Tsouros, Stamatia Bibi, and Panagiotis G. Sarigiannidis. "A Review on UAV-Based Applications for Precision Agriculture." In: *Information* 10.11 (2019). ISSN: 2078-2489. DOI: 10.3390/info10110349. URL: https://www.mdpi.com/2078-2489/10/11/349.

[2] Dario Floreano and Robert J. Wood. "Science, technology and the future of small autonomous drones." In: *Nature* 521.7553 (2015), pp. 460–466. URL: https://EconPapers.repec.org/RePEc:nat:nature:v:521:y:2015:i:7553:d:10.1038_nature14542.

[3] Hazim Shakhatreh et al. "Unmanned Aerial Vehicles (UAVs): A Survey on Civil Applications and Key Research Challenges." In: *IEEE Access* 7 (Apr. 2019), pp. 48572–48634. DOI: 10.1109/ACCESS.2019.2909530.

[4] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. "Deep learning." In: *Nature* 521.7553 (May 2015), pp. 436–444. ISSN: 1476-4687. DOI: 10.1038/nature14539. URL: https://doi.org/10.1038/nature14539.

[5] Liu Jian et al. "Understanding of Channel State Information." In: *Smart Wireless Sensing*. Springer, 2021, pp. 19–41. URL: https://link.springer.com/chapter/10.1007/978-981-16-5658-3_2.

[6] Q Li, P Wang, and Y Du. "WiFi gesture recognition method based on data enhancement." In: *Fourth International Conference on Wireless and Mobile Communications*. SPIE, 2024. URL: https://www.spiedigitallibrary.org/conference-proceedings-of-spie/13176/131762C/WiFi-gesture-recognition-method-based-on-data-enhancement/10.1117/12.3029330.short.

[7] Y Wang. "High Accuracy WiFi Sensing for Vital Sign Detection with Multi-Task Contrastive Learning." PhD thesis. University of Sydney, 2024. URL: https://ses.library.usyd.edu.au/handle/2123/32568.

[8] W Li et al. "WiFi-CSI Difference Paradigm: Achieving Efficient Doppler Speed Estimation for Passive Tracking." In: *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies* (2024). URL: https://dl.acm.org/doi/abs/10.1145/3659608.

[9] Y Mao et al. "Wi-Cro: WiFi-based Cross Domain Activity Recognition via Modified GAN." In: *IEEE Transactions on Wireless Communications* (2024). URL: https://ieeexplore.ieee.org/abstract/document/10538035/.

[10] F Wu et al. "Multiscale Low-Frequency Memory Network for Improved Feature Extraction in Convolutional Neural Networks." In: *Proceedings of the AAAI Conference on Artificial Intelligence* (2024). URL: https://ojs.aaai.org/index.php/AAAI/article/view/28411/28803.

[11] B Maurya and S Hiranwal. "Mathematical Modeling and Statistical Exploration of Residual Computing Based Convolutional Neural Network Based Classifier for Complex Image Demarcation." In: *Communications on Applied Network Analysis* (2024). URL: https://internationalpubls.com/index.php/cana/article/download/441/392.

[12] A Jamil. "DepGraphRL: A Deep Reinforcement Learning-Driven, Energy-Aware, Dependency Graph-Based Pruning Framework for Edge-IoT Devices." In: *Gnosis Library, University of Cyprus* (2024). URL: https://gnosis.library.ucy.ac.cy/bitstream/handle/7/66138/Asfa_Jamil_2024_secured.pdf?sequence=4.

[13] Jian Liu et al. "Wireless Sensing for Human Activity: A Survey." In: *IEEE Communications Surveys & Tutorials* 22.3 (2020), pp. 1629–1648. DOI: 10.1109/COMST.2019.2934489.

[14] M Krovvidi. "Activity Detection Using RF and Sensor Data Fusion." In: *kilthub.cmu.edu* (2024). URL: https://kilthub.cmu.edu/ndownloader/files/46240501.

[15] R Candell et al. "Wireless Deployment Challenges in Construction: A 5G Strategy." In: *tsapps.nist.gov* (2024). URL: https://tsapps.nist.gov/publication/get_pdf.cfm?pub_id=957228.

[16] SS Yadav et al. "tinyRadar: LSTM-based Real-time Multi-target Human Activity Recognition for Edge Computing." In: *labs.dese.iisc.ac.in* (2024). URL: https://labs.dese.iisc.ac.in/neuronics/wp-content/uploads/sites/16/2024/05/Human_activity_Recognition_LSTM_ISCAS_2024-1.pdf.

[17] J Tian, P Mercier, and C Paolini. "Ultra low-power, wearable, accelerated shallow-learning fall detection for elderly at-risk persons." In: *Smart Health* (2024). URL: https://www.sciencedirect.com/science/article/pii/S2352648324000540.

[18] R Mazzieri, J Pegoraro, and M Rossi. "Attention-Refined Unrolling for Sparse Sequential micro-Doppler Reconstruction." In: *IEEE Journal of Selected Topics in Signal Processing* (2024). URL: https://ieeexplore.ieee.org/abstract/document/10543012/.

[19] HAD Nguyen, TH Le, and QP Ha. "Deep learning for construction emission monitoring with low-cost sensor network." In: *Proceedings of the 40th*. 2023. URL: https://opus.lib.uts.edu.au/bitstream/10453/171517/4/Deep%20learning%20for%20construction%20emission%20monitoring%20with%20low-cost%20sensor%20network.pdf.

[20] R Santhosh et al. "An Intelligent Robust One Dimensional HAR-CNN Model for Human Activity Recognition using Wearable Sensor Data." In: *ResearchGate* (2023). URL: https://www.researchgate.net/profile/Deepan-Perumal/publication/369589570_An_Intelligent_Robust_One_Dimensional_HAR-CNN_Model_for_Human_Activity_Recognition_using_Wearable_Sensor_Data/links/658ea3443c472d2e8e9c An-Intelligent-Robust-One-Dimensional-HAR-CNN-Model-for-Human-Activity-Recognition-using-Wearable-Sensor-Data.pdf.

[21] J Rostovski, MH Ahmadilivani, and A Krivošei. "Real-Time Gait Anomaly Detection Using 1D-CNN and LSTM." In: *Lecture Notes in Networks and Systems* (2024). URL: https://link.springer.com/chapter/10.1007/978-3-031-59091-7_17.

[22] RRX Lau. "Investigation into wireless sensing using channel state information." In: *Nanyang Technological University* (2024). URL: https://dr.ntu.edu.sg/handle/10356/175041.

[23] *ThinkPad T460s | 14" Enterprise-Ready Ultrabook; | Lenovo US — lenovo.com*. https://www.lenovo.com/us/en/p/laptops/thinkpad/thinkpadt/thinkpad-t460s/22tp2tt460s. [Accessed 31-05-2024].

[24] Joel Santo Domingo. *Lenovo ThinkPad T460s Review — pcmag.com*. https://www.pcmag.com/reviews/lenovo-thinkpad-t460s. [Accessed 31-05-2024]. 2016.

[25] *Acer Aspire 5*. https://www.acer.com/us-en/laptops/aspire/aspire-5-amd/pdp/NX.K82AA.001. [Accessed 21-05-2024].

[26] *GNU Radio — gnuradio.org*. https://www.gnuradio.org/. [Accessed 21-05-2024].

[27] *Canonical releases Ubuntu 24.04 LTS Noble Numbat | Ubuntu — ubuntu.com*. https://ubuntu.com/blog/canonical-releases-ubuntu-24-04-noble-numbat. [Accessed 21-05-2024].

[28] *LTS - Ubuntu Wiki — wiki.ubuntu.com*. https://wiki.ubuntu.com/LTS. [Accessed 07-06-2024].

[29] *ubuntu.com*. https://ubuntu.com/desktop. [Accessed 07-06-2024].

[30] *balenaEtcher - Flash OS images to SD cards & USB drives — etcher.balena.io*. https://etcher.balena.io/. [Accessed 21-05-2024].

[31] *USRP E312 Datasheet*. https://www.ettus.com/wp-content/uploads/2019/01/USRP_E312_Datasheet.pdf. [Accessed 26-05-2024].

[32] *E310/E312 - Ettus Knowledge Base — kb.ettus.com*. https://kb.ettus.com/E310/E312. [Accessed 26-05-2024].

[33] Ettus Research. *UHD - USRP Hardware Driver*. https://www.ettus.com/all-products/uhd/. Accessed: 2024-05-26. 2024.

[34] USRP Users. *USRP Users Mailing List*. http://lists.ettus.com/mailman/listinfo/usrp-users_lists.ettus.com. Accessed: 2024-05-26. 2024.

[35] a National Instruments Brand Ettus Research. *VERT400 Antenna — ettus.com*. https://www.ettus.com/all-products/vert400/. [Accessed 06-06-2024].

[36] DJI. *DJI Mini SE*. https://www.dji.com/mini-se. Accessed: 2024-06-07. 2024.

[37] DJI. *DJI Mini 2*. https://www.dji.com/mini-2. Accessed: 2024-06-07. 2024.

[38] Robolink. *CoDrone Edu*. https://www.robolink.com/codrone-edu/. Accessed: 2024-06-07. 2024.

[39] *ubuntu.com*. https://ubuntu.com/tutorials/install-ubuntu-desktop. [Accessed 07-06-2024].

[40] *Certified laptops | Ubuntu — ubuntu.com*. https://ubuntu.com/certified/laptops. [Accessed 26-05-2024].

[41] N Ali, M Imran, and H Abbasi. "Performance Analysis of Bluetooth and Wi-Fi for Wireless Sensing." In: *Journal of Wireless Communications* (2018). URL: https://example.com/ali2018.

[42] L Zhang, Y Wang, and T Li. "Limitations and Challenges of Wi-Fi and Bluetooth Sensing." In: *International Journal of Wireless Networks* (2019). URL: https://example.com/zhang2019.

[43] R RajBharath, S Bhalamourale, and D Saaranathan. "Real Time Fall Detection for Geriatric Risk Assessment using CNNs." In: *Journal of Biomedical Systems* (2024). URL: https://www.afjbs.com/uploads/paper/a1fea3ec048eed08b7a570a69411b716.pdf.

[44] L Gong and Y Chen. "Machine Learning-enhanced IoT and Wireless Sensor Networks for Predictive Analysis and Maintenance in Wind Turbine Systems." In: *International Journal of Intelligent Networks* (2024). URL: https://www.sciencedirect.com/science/article/pii/S2666603024000083.

[45] A Tahat, TA Edwan, and MB Dababseh. "A Versatile Machine Learning-Based Vehicle-to-Vehicle Connectivity Model." In: *IEEE Communications Magazine* (2023). URL: https://ieeexplore.ieee.org/abstract/document/10187740/.

[46] A Andrianingsih, EP Wibowo, and IKA Enriko. "Experimental Visualization of the LoRaWAN Variable Correlation in Jakarta." In: *IEEE Communications Letters* (2024). URL: https://ieeexplore.ieee.org/abstract/document/10422795/.

[47] T Matsunaga, I Arai, and Y Atarashi. "Developing Dense Wireless Signal and Magnetic Field Mapping Tool." In: *IEEE Communications Letters* (2024). URL: https://ieeexplore.ieee.org/abstract/document/10503421/.

[48] T Prihantoro, D Perdana, and AI Irawan. "Performance Analysis Antares Long Range Technology with Mobile Application Visualization." In: *International Conference on Wireless and Mobile Communications*. 2023. URL: https://ieeexplore.ieee.org/abstract/document/10335205/.

Appendix A

5G System Code

A.1 OFDM Transmitter and Receiver

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

#
# SPDX-License-Identifier: GPL-3.0
#
# GNU Radio Python Flow Graph
# Title: OFDM Transmitter
# Description: Transmit a pre-defined signal (a complex sine) as OFDM ...
    packets.
# GNU Radio version: v3.11.0.0git-717-g72e21b54

from PyQt5 import Qt
from gnuradio import qtgui
from PyQt5 import QtCore
from gnuradio import analog
from gnuradio import blocks
from gnuradio import digital
from gnuradio import gr
from gnuradio.filter import firdes
from gnuradio.fft import window
import sys
import signal
from PyQt5 import Qt
from argparse import ArgumentParser
from gnuradio.eng_arg import eng_float, intx
from gnuradio import eng_notation
from gnuradio import uhd
import time
from gnuradio.digital.utils import tagged_streams
import sip
```

```
31
32
33 class OFDM_Tx(gr.top_block, Qt.QWidget):
34
35     def __init__(self):
36         gr.top_block.__init__(self, "OFDM Transmitter", ...
    catch_exceptions=True)
37         Qt.QWidget.__init__(self)
38         self.setWindowTitle("OFDM Transmitter")
39         qtgui.util.check_set_qss()
40         try:
41             self.setWindowIcon(Qt.QIcon.fromTheme('gnuradio-grc'))
42         except BaseException as exc:
43             print(f"Qt GUI: Could not set Icon: {str(exc)}", file=sys....
    stderr)
44         self.top_scroll_layout = Qt.QVBoxLayout()
45         self.setLayout(self.top_scroll_layout)
46         self.top_scroll = Qt.QScrollArea()
47         self.top_scroll.setFrameStyle(Qt.QFrame.NoFrame)
48         self.top_scroll_layout.addWidget(self.top_scroll)
49         self.top_scroll.setWidgetResizable(True)
50         self.top_widget = Qt.QWidget()
51         self.top_scroll.setWidget(self.top_widget)
52         self.top_layout = Qt.QVBoxLayout(self.top_widget)
53         self.top_grid_layout = Qt.QGridLayout()
54         self.top_layout.addLayout(self.top_grid_layout)
55
56         self.settings = Qt.QSettings("GNU Radio", "OFDM_Tx")
57
58         try:
59             geometry = self.settings.value("geometry")
60             if geometry:
61                 self.restoreGeometry(geometry)
62         except BaseException as exc:
63             print(f"Qt GUI: Could not restore geometry: {str(exc)}", ...
    file=sys.stderr)
64
65         ##################################################
66         # Variables
67         ##################################################
68         self.samp_rate = samp_rate = 1.5e6
69         self.packet_length_tag_key = packet_length_tag_key = "...
    packet_len"
70         self.packet_len = packet_len = 50
71         self.noise_voltage = noise_voltage = 0.1
72         self.len_tag_key = len_tag_key = "packet_len"
73         self.freq_offset = freq_offset = 0
74         self.fft_len = fft_len = 64
75         self.center_freq = center_freq = 980e6
```

```
76         self.band_width = band_width = 7.5e5
77 
78         ##################################################
79         # Blocks
80         ##################################################
81 
82         self.uhd_usrp_sink_0 = uhd.usrp_sink(
83             ",".join(("addr=128.39.200.106", '')),
84             uhd.stream_args(
85                 cpu_format="fc32",
86                 args='',
87                 channels=list(range(0, 1)),
88             ),
89             "",
90         )
91         self.uhd_usrp_sink_0.set_samp_rate(samp_rate)
92         self.uhd_usrp_sink_0.set_time_unknown_pps(uhd.time_spec(0))
93 
94         self.uhd_usrp_sink_0.set_center_freq(center_freq, 0)
95         self.uhd_usrp_sink_0.set_antenna("TX/RX", 0)
96         self.uhd_usrp_sink_0.set_bandwidth(1e6, 0)
97         self.uhd_usrp_sink_0.set_gain(50, 0)
98         self.qtgui_waterfall_sink_x_0 = qtgui.waterfall_sink_c(
99             1024,  # size
100            window.WIN_HAMMING,  # wintype
101            center_freq,  # fc
102            band_width,  # bw
103            "",  # name
104            1,  # number of inputs
105            None,  # parent
106        )
107        self.qtgui_waterfall_sink_x_0.set_update_time(0.10)
108        self.qtgui_waterfall_sink_x_0.enable_grid(True)
109        self.qtgui_waterfall_sink_x_0.enable_axis_labels(True)
110
111        labels = ['', '', '', '', '', '', '', '', '', '']
112        colors = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
113        alphas = [1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0]
114
115        for i in range(1):
116            if len(labels[i]) == 0:
117                self.qtgui_waterfall_sink_x_0.set_line_label(i, "Data ...
    {0}".format(i))
118            else:
119                self.qtgui_waterfall_sink_x_0.set_line_label(i, labels[...
    i])
120            self.qtgui_waterfall_sink_x_0.set_color_map(i, colors[i])
121            self.qtgui_waterfall_sink_x_0.set_line_alpha(i, alphas[i])
122
```

```
        self.qtgui_waterfall_sink_x_0.set_intensity_range(-140, 10)

        self._qtgui_waterfall_sink_x_0_win = sip.wrapinstance(
            self.qtgui_waterfall_sink_x_0.qwidget(), Qt.QWidget
        )

        self.top_layout.addWidget(self._qtgui_waterfall_sink_x_0_win)
        self.qtgui_time_sink_x_0_0_1 = qtgui.time_sink_c(
            1024,  # size
            samp_rate,  # samp_rate
            'Transmitter Input',  # name
            1,  # number of inputs
            None,  # parent
        )
        self.qtgui_time_sink_x_0_0_1.set_update_time(0.10)
        self.qtgui_time_sink_x_0_0_1.set_y_axis(-1, 1)

        self.qtgui_time_sink_x_0_0_1.set_y_label('Amplitude', "")

        self.qtgui_time_sink_x_0_0_1.enable_tags(True)
        self.qtgui_time_sink_x_0_0_1.set_trigger_mode(
            qtgui.TRIG_MODE_FREE, qtgui.TRIG_SLOPE_POS, 0.0, 0, 0, ""
        )
        self.qtgui_time_sink_x_0_0_1.enable_autoscale(True)
        self.qtgui_time_sink_x_0_0_1.enable_grid(False)
        self.qtgui_time_sink_x_0_0_1.enable_axis_labels(True)
        self.qtgui_time_sink_x_0_0_1.enable_control_panel(True)
        self.qtgui_time_sink_x_0_0_1.enable_stem_plot(False)

        labels = ['', '', '', '', '', '', '', '', '', '']
        widths = [1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
        colors = [
            'blue',
            'red',
            'green',
            'black',
            'cyan',
            'magenta',
            'yellow',
            'dark red',
            'dark green',
            'dark blue',
        ]
        alphas = [1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0]
        styles = [1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
        markers = [-1, -1, -1, -1, -1, -1, -1, -1, -1, -1]

        for i in range(2):
            if len(labels[i]) == 0:
```

```
172                 if i % 2 == 0:
173                     self.qtgui_time_sink_x_0_0_1.set_line_label(
174                         i, "Re{{Data {0}}}".format(i / 2)
175                     )
176                 else:
177                     self.qtgui_time_sink_x_0_0_1.set_line_label(
178                         i, "Im{{Data {0}}}".format(i / 2)
179                     )
180             else:
181                 self.qtgui_time_sink_x_0_0_1.set_line_label(i, labels[i...
    ])
182             self.qtgui_time_sink_x_0_0_1.set_line_width(i, widths[i])
183             self.qtgui_time_sink_x_0_0_1.set_line_color(i, colors[i])
184             self.qtgui_time_sink_x_0_0_1.set_line_style(i, styles[i])
185             self.qtgui_time_sink_x_0_0_1.set_line_marker(i, markers[i])
186             self.qtgui_time_sink_x_0_0_1.set_line_alpha(i, alphas[i])
187 
188         self._qtgui_time_sink_x_0_0_1_win = sip.wrapinstance(
189             self.qtgui_time_sink_x_0_0_1.qwidget(), Qt.QWidget
190         )
191         self.top_layout.addWidget(self._qtgui_time_sink_x_0_0_1_win)
192         self.qtgui_time_sink_x_0 = qtgui.time_sink_f(
193             1024,  # size
194             samp_rate,  # samp_rate
195             "Data",  # name
196             1,  # number of inputs
197             None,  # parent
198         )
199         self.qtgui_time_sink_x_0.set_update_time(0.10)
200         self.qtgui_time_sink_x_0.set_y_axis(-1, 1)
201 
202         self.qtgui_time_sink_x_0.set_y_label('Amplitude', "")
203 
204         self.qtgui_time_sink_x_0.enable_tags(True)
205         self.qtgui_time_sink_x_0.set_trigger_mode(
206             qtgui.TRIG_MODE_FREE, qtgui.TRIG_SLOPE_POS, 0.0, 0, 0, ""
207         )
208         self.qtgui_time_sink_x_0.enable_autoscale(True)
209         self.qtgui_time_sink_x_0.enable_grid(True)
210         self.qtgui_time_sink_x_0.enable_axis_labels(True)
211         self.qtgui_time_sink_x_0.enable_control_panel(True)
212         self.qtgui_time_sink_x_0.enable_stem_plot(False)
213 
214         labels = [
215             'Signal 1',
216             'Signal 2',
217             'Signal 3',
218             'Signal 4',
219             'Signal 5',
```

```
220             'Signal 6',
221             'Signal 7',
222             'Signal 8',
223             'Signal 9',
224             'Signal 10',
225         ]
226         widths = [1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
227         colors = [
228             'blue',
229             'red',
230             'green',
231             'black',
232             'cyan',
233             'magenta',
234             'yellow',
235             'dark red',
236             'dark green',
237             'dark blue',
238         ]
239         alphas = [1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0]
240         styles = [1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
241         markers = [-1, -1, -1, -1, -1, -1, -1, -1, -1, -1]
242 
243         for i in range(1):
244             if len(labels[i]) == 0:
245                 self.qtgui_time_sink_x_0.set_line_label(i, "Data {0}"....
    format(i))
246             else:
247                 self.qtgui_time_sink_x_0.set_line_label(i, labels[i])
248             self.qtgui_time_sink_x_0.set_line_width(i, widths[i])
249             self.qtgui_time_sink_x_0.set_line_color(i, colors[i])
250             self.qtgui_time_sink_x_0.set_line_style(i, styles[i])
251             self.qtgui_time_sink_x_0.set_line_marker(i, markers[i])
252             self.qtgui_time_sink_x_0.set_line_alpha(i, alphas[i])
253 
254         self._qtgui_time_sink_x_0_win = sip.wrapinstance(
255             self.qtgui_time_sink_x_0.qwidget(), Qt.QWidget
256         )
257         self.top_layout.addWidget(self._qtgui_time_sink_x_0_win)
258         self.qtgui_freq_sink_x_0_0 = qtgui.freq_sink_c(
259             1024,  # size
260             window.WIN_HAMMING,  # wintype
261             center_freq,  # fc
262             band_width,  # bw
263             'Rx Spectrum1',  # name
264             1,
265             None,  # parent
266         )
267         self.qtgui_freq_sink_x_0_0.set_update_time(0.10)
```

```
268         self.qtgui_freq_sink_x_0_0.set_y_axis((-140), 10)
269         self.qtgui_freq_sink_x_0_0.set_y_label('Relative Gain', 'dB')
270         self.qtgui_freq_sink_x_0_0.set_trigger_mode(qtgui....
    TRIG_MODE_FREE, 0.0, 0, "")
271         self.qtgui_freq_sink_x_0_0.enable_autoscale(True)
272         self.qtgui_freq_sink_x_0_0.enable_grid(True)
273         self.qtgui_freq_sink_x_0_0.set_fft_average(1.0)
274         self.qtgui_freq_sink_x_0_0.enable_axis_labels(True)
275         self.qtgui_freq_sink_x_0_0.enable_control_panel(True)
276         self.qtgui_freq_sink_x_0_0.set_fft_window_normalized(False)
277 
278         labels = ['Rx Spectrum', '', '', '', '', '', '', '', '', '']
279         widths = [1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
280         colors = [
281             "blue",
282             "red",
283             "green",
284             "black",
285             "cyan",
286             "magenta",
287             "yellow",
288             "dark red",
289             "dark green",
290             "dark blue",
291         ]
292         alphas = [1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0]
293 
294         for i in range(1):
295             if len(labels[i]) == 0:
296                 self.qtgui_freq_sink_x_0_0.set_line_label(i, "Data {0}"...
    .format(i))
297             else:
298                 self.qtgui_freq_sink_x_0_0.set_line_label(i, labels[i])
299             self.qtgui_freq_sink_x_0_0.set_line_width(i, widths[i])
300             self.qtgui_freq_sink_x_0_0.set_line_color(i, colors[i])
301             self.qtgui_freq_sink_x_0_0.set_line_alpha(i, alphas[i])
302 
303         self._qtgui_freq_sink_x_0_0_win = sip.wrapinstance(
304             self.qtgui_freq_sink_x_0_0.qwidget(), Qt.QWidget
305         )
306         self.top_layout.addWidget(self._qtgui_freq_sink_x_0_0_win)
307         self._noise_voltage_range = qtgui.Range(0, 1, 0.01, 0.1, 200)
308         self._noise_voltage_win = qtgui.RangeWidget(
309             self._noise_voltage_range,
310             self.set_noise_voltage,
311             "Noise Amplitude",
312             "counter_slider",
313             float,
314             QtCore.Qt.Horizontal,
```

```
315         )
316         self.top_layout.addWidget(self._noise_voltage_win)
317         self._freq_offset_range = qtgui.Range(-3, 3, 0.01, 0, 200)
318         self._freq_offset_win = qtgui.RangeWidget(
319             self._freq_offset_range,
320             self.set_freq_offset,
321             "Frequency Offset (Multiples of Sub-carrier spacing)",
322             "counter_slider",
323             float,
324             QtCore.Qt.Horizontal,
325         )
326         self.top_layout.addWidget(self._freq_offset_win)
327         self.digital_ofdm_tx_0 = digital.ofdm_tx(
328             fft_len=fft_len,
329             cp_len=(fft_len // 4),
330             packet_length_tag_key=len_tag_key,
331             occupied_carriers=((-4, -3, -2, -1, 1, 2, 3, 4),),
332             pilot_carriers=((-6, -5, 5, 6),),
333             pilot_symbols=((-1, 1, -1, 1),),
334             sync_word1=None,
335             sync_word2=None,
336             bps_header=1,
337             bps_payload=2,
338             rolloff=0,
339             debug_log=True,
340             scramble_bits=False,
341         )
342         self.blocks_uchar_to_float_0 = blocks.uchar_to_float()
343         self.blocks_stream_to_tagged_stream_0 = blocks....
    stream_to_tagged_stream(
344             gr.sizeof_char, 1, packet_len, len_tag_key
345         )
346         self.blocks_multiply_const_vxx_0_0 = blocks.multiply_const_cc...
    (0.05)
347         self.blocks_file_sink_0_0 = blocks.file_sink(
348             gr.sizeof_char * 1,
349             '/home/acps/Documents/data/Velocity_Track_OFDM_Tx',
350             False,
351         )
352         self.blocks_file_sink_0_0.set_unbuffered(False)
353         self.analog_const_source_x_0 = analog.sig_source_b(
354             0, analog.GR_CONST_WAVE, 0, 0, 50
355         )
356 
357         ##################################################
358         # Connections
359         ##################################################
360         self.connect(
361             (self.analog_const_source_x_0, 0),
```

```
362            (self.blocks_stream_to_tagged_stream_0, 0),
363        )
364        self.connect(
365            (self.blocks_multiply_const_vxx_0_0, 0), (self....
    qtgui_freq_sink_x_0_0, 0)
366        )
367        self.connect(
368            (self.blocks_multiply_const_vxx_0_0, 0), (self....
    qtgui_time_sink_x_0_0_1, 0)
369        )
370        self.connect(
371            (self.blocks_multiply_const_vxx_0_0, 0), (self....
    qtgui_waterfall_sink_x_0, 0)
372        )
373        self.connect((self.blocks_multiply_const_vxx_0_0, 0), (self....
    uhd_usrp_sink_0, 0))
374        self.connect(
375            (self.blocks_stream_to_tagged_stream_0, 0), (self....
    blocks_file_sink_0_0, 0)
376        )
377        self.connect(
378            (self.blocks_stream_to_tagged_stream_0, 0),
379            (self.blocks_uchar_to_float_0, 0),
380        )
381        self.connect(
382            (self.blocks_stream_to_tagged_stream_0, 0), (self....
    digital_ofdm_tx_0, 0)
383        )
384        self.connect((self.blocks_uchar_to_float_0, 0), (self....
    qtgui_time_sink_x_0, 0))
385        self.connect(
386            (self.digital_ofdm_tx_0, 0), (self....
    blocks_multiply_const_vxx_0_0, 0)
387        )
388
389    def closeEvent(self, event):
390        self.settings = Qt.QSettings("GNU Radio", "OFDM_Tx")
391        self.settings.setValue("geometry", self.saveGeometry())
392        self.stop()
393        self.wait()
394
395        event.accept()
396
397    def get_samp_rate(self):
398        return self.samp_rate
399
400    def set_samp_rate(self, samp_rate):
401        self.samp_rate = samp_rate
402        self.qtgui_time_sink_x_0.set_samp_rate(self.samp_rate)
```

```
403         self.qtgui_time_sink_x_0_0_1.set_samp_rate(self.samp_rate)
404         self.uhd_usrp_sink_0.set_samp_rate(self.samp_rate)
405 
406     def get_packet_length_tag_key(self):
407         return self.packet_length_tag_key
408 
409     def set_packet_length_tag_key(self, packet_length_tag_key):
410         self.packet_length_tag_key = packet_length_tag_key
411 
412     def get_packet_len(self):
413         return self.packet_len
414 
415     def set_packet_len(self, packet_len):
416         self.packet_len = packet_len
417         self.blocks_stream_to_tagged_stream_0.set_packet_len(self....
    packet_len)
418         self.blocks_stream_to_tagged_stream_0.set_packet_len_pmt(self....
    packet_len)
419 
420     def get_noise_voltage(self):
421         return self.noise_voltage
422 
423     def set_noise_voltage(self, noise_voltage):
424         self.noise_voltage = noise_voltage
425 
426     def get_len_tag_key(self):
427         return self.len_tag_key
428 
429     def set_len_tag_key(self, len_tag_key):
430         self.len_tag_key = len_tag_key
431 
432     def get_freq_offset(self):
433         return self.freq_offset
434 
435     def set_freq_offset(self, freq_offset):
436         self.freq_offset = freq_offset
437 
438     def get_fft_len(self):
439         return self.fft_len
440 
441     def set_fft_len(self, fft_len):
442         self.fft_len = fft_len
443 
444     def get_center_freq(self):
445         return self.center_freq
446 
447     def set_center_freq(self, center_freq):
448         self.center_freq = center_freq
449         self.qtgui_freq_sink_x_0_0.set_frequency_range(
```

```
450                 self.center_freq, self.band_width
451             )
452             self.qtgui_waterfall_sink_x_0.set_frequency_range(
453                 self.center_freq, self.band_width
454             )
455             self.uhd_usrp_sink_0.set_center_freq(self.center_freq, 0)
456 
457         def get_band_width(self):
458             return self.band_width
459 
460         def set_band_width(self, band_width):
461             self.band_width = band_width
462             self.qtgui_freq_sink_x_0_0.set_frequency_range(
463                 self.center_freq, self.band_width
464             )
465             self.qtgui_waterfall_sink_x_0.set_frequency_range(
466                 self.center_freq, self.band_width
467             )
468 
469 
470 def main(top_block_cls=OFDM_Tx, options=None):
471 
472     qapp = Qt.QApplication(sys.argv)
473 
474     tb = top_block_cls()
475 
476     tb.start()
477 
478     tb.show()
479 
480     def sig_handler(sig=None, frame=None):
481         tb.stop()
482         tb.wait()
483 
484         Qt.QApplication.quit()
485 
486     signal.signal(signal.SIGINT, sig_handler)
487     signal.signal(signal.SIGTERM, sig_handler)
488 
489     timer = Qt.QTimer()
490     timer.start(500)
491     timer.timeout.connect(lambda: None)
492 
493     qapp.exec_()
494 
495 
496 if __name__ == '__main__':
497     main()
```

Listing A.1: OFDM Transmitter

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

#
# SPDX-License-Identifier: GPL-3.0
#
# GNU Radio Python Flow Graph
# Title: OFDM receiver
# Description: Transmit a pre-defined signal (a complex sine) as OFDM ...
    packets.
# GNU Radio version: v3.11.0.0git-717-g72e21b54

from PyQt5 import Qt
from gnuradio import qtgui
from PyQt5 import QtCore
from gnuradio import blocks
from gnuradio import digital
from gnuradio import gr
from gnuradio.filter import firdes
from gnuradio.fft import window
import sys
import signal
from PyQt5 import Qt
from argparse import ArgumentParser
from gnuradio.eng_arg import eng_float, intx
from gnuradio import eng_notation
from gnuradio import uhd
import time
from gnuradio.digital.utils import tagged_streams
import ofdm_Rx_epy_block_0 as epy_block_0  # embedded python block
import sip



class ofdm_Rx(gr.top_block, Qt.QWidget):

    def __init__(self):
        gr.top_block.__init__(self, "OFDM receiver", catch_exceptions=...
    True)
        Qt.QWidget.__init__(self)
        self.setWindowTitle("OFDM receiver")
        qtgui.util.check_set_qss()
        try:
            self.setWindowIcon(Qt.QIcon.fromTheme('gnuradio-grc'))
        except BaseException as exc:
            print(f"Qt GUI: Could not set Icon: {str(exc)}", file=sys....
    stderr)
        self.top_scroll_layout = Qt.QVBoxLayout()
        self.setLayout(self.top_scroll_layout)
```

```
47         self.top_scroll = Qt.QScrollArea()
48         self.top_scroll.setFrameStyle(Qt.QFrame.NoFrame)
49         self.top_scroll_layout.addWidget(self.top_scroll)
50         self.top_scroll.setWidgetResizable(True)
51         self.top_widget = Qt.QWidget()
52         self.top_scroll.setWidget(self.top_widget)
53         self.top_layout = Qt.QVBoxLayout(self.top_widget)
54         self.top_grid_layout = Qt.QGridLayout()
55         self.top_layout.addLayout(self.top_grid_layout)
56 
57         self.settings = Qt.QSettings("GNU Radio", "ofdm_Rx")
58 
59         try:
60             geometry = self.settings.value("geometry")
61             if geometry:
62                 self.restoreGeometry(geometry)
63         except BaseException as exc:
64             print(f"Qt GUI: Could not restore geometry: {str(exc)}", ...
    file=sys.stderr)
65 
66         ##################################################
67         # Variables
68         ##################################################
69         self.samp_rate = samp_rate = 1.5e6
70         self.packet_length_tag_key = packet_length_tag_key = "...
    packet_len"
71         self.packet_len = packet_len = 50
72         self.noise_voltage = noise_voltage = 0.1
73         self.len_tag_key = len_tag_key = "packet_len"
74         self.freq_offset = freq_offset = 0
75         self.fft_len = fft_len = 64
76         self.center_freq = center_freq = 980e6
77         self.band_width = band_width = 75e4
78 
79         ##################################################
80         # Blocks
81         ##################################################
82 
83         self.uhd_usrp_source_0 = uhd.usrp_source(
84             ",".join(("addr=128.39.200.112", '')),
85             uhd.stream_args(
86                 cpu_format="fc32",
87                 args='',
88                 channels=list(range(0,1)),
89             ),
90         )
91         self.uhd_usrp_source_0.set_samp_rate(samp_rate)
92         self.uhd_usrp_source_0.set_time_unknown_pps(uhd.time_spec(0))
93 
```

```
94          self.uhd_usrp_source_0.set_center_freq(center_freq, 0)
95          self.uhd_usrp_source_0.set_antenna("RX2", 0)
96          self.uhd_usrp_source_0.set_bandwidth(1e6, 0)
97          self.uhd_usrp_source_0.set_gain(70, 0)
98          self.qtgui_waterfall_sink_x_0_0 = qtgui.waterfall_sink_c(
99              1024, #size
100             window.WIN_HAMMING, #wintype
101             center_freq, #fc
102             band_width, #bw
103             "CSI Specto", #name
104             1, #number of inputs
105             None # parent
106         )
107         self.qtgui_waterfall_sink_x_0_0.set_update_time(0.10)
108         self.qtgui_waterfall_sink_x_0_0.enable_grid(True)
109         self.qtgui_waterfall_sink_x_0_0.enable_axis_labels(True)
110
111
112
113         labels = ['', '', '', '', '',
114                   '', '', '', '', '']
115         colors = [0, 0, 0, 0, 0,
116                   0, 0, 0, 0, 0]
117         alphas = [1.0, 1.0, 1.0, 1.0, 1.0,
118                   1.0, 1.0, 1.0, 1.0, 1.0]
119
120         for i in range(1):
121             if len(labels[i]) == 0:
122                 self.qtgui_waterfall_sink_x_0_0.set_line_label(i, "Data...
     {0}".format(i))
123             else:
124                 self.qtgui_waterfall_sink_x_0_0.set_line_label(i, ...
   labels[i])
125             self.qtgui_waterfall_sink_x_0_0.set_color_map(i, colors[i])
126             self.qtgui_waterfall_sink_x_0_0.set_line_alpha(i, alphas[i...
   ])
127
128         self.qtgui_waterfall_sink_x_0_0.set_intensity_range(-140, 10)
129
130         self._qtgui_waterfall_sink_x_0_0_win = sip.wrapinstance(self....
   qtgui_waterfall_sink_x_0_0.qwidget(), Qt.QWidget)
131
132         self.top_layout.addWidget(self._qtgui_waterfall_sink_x_0_0_win)
133         self.qtgui_waterfall_sink_x_0 = qtgui.waterfall_sink_f(
134             1024, #size
135             window.WIN_HAMMING, #wintype
136             center_freq, #fc
137             band_width, #bw
138             "RSSI Spectogram", #name
```

```
139             1, #number of inputs
140             None # parent
141         )
142         self.qtgui_waterfall_sink_x_0.set_update_time(0.10)
143         self.qtgui_waterfall_sink_x_0.enable_grid(True)
144         self.qtgui_waterfall_sink_x_0.enable_axis_labels(True)
145 
146 
147         self.qtgui_waterfall_sink_x_0.set_plot_pos_half(not True)
148 
149         labels = ['', '', '', '', '',
150                   '', '', '', '', '']
151         colors = [0, 0, 0, 0, 0,
152                   0, 0, 0, 0, 0]
153         alphas = [1.0, 1.0, 1.0, 1.0, 1.0,
154                   1.0, 1.0, 1.0, 1.0, 1.0]
155 
156         for i in range(1):
157             if len(labels[i]) == 0:
158                 self.qtgui_waterfall_sink_x_0.set_line_label(i, "Data ...
    {0}".format(i))
159             else:
160                 self.qtgui_waterfall_sink_x_0.set_line_label(i, labels[...
    i])
161             self.qtgui_waterfall_sink_x_0.set_color_map(i, colors[i])
162             self.qtgui_waterfall_sink_x_0.set_line_alpha(i, alphas[i])
163 
164         self.qtgui_waterfall_sink_x_0.set_intensity_range(-140, 10)
165 
166         self._qtgui_waterfall_sink_x_0_win = sip.wrapinstance(self....
    qtgui_waterfall_sink_x_0.qwidget(), Qt.QWidget)
167 
168         self.top_layout.addWidget(self._qtgui_waterfall_sink_x_0_win)
169         self.qtgui_time_sink_x_1 = qtgui.time_sink_f(
170             1024, #size
171             samp_rate, #samp_rate
172             "RSSI output", #name
173             1, #number of inputs
174             None # parent
175         )
176         self.qtgui_time_sink_x_1.set_update_time(0.10)
177         self.qtgui_time_sink_x_1.set_y_axis(-1, 1)
178 
179         self.qtgui_time_sink_x_1.set_y_label('Amplitude', "")
180 
181         self.qtgui_time_sink_x_1.enable_tags(True)
182         self.qtgui_time_sink_x_1.set_trigger_mode(qtgui.TRIG_MODE_FREE,...
     qtgui.TRIG_SLOPE_POS, 0.0, 0, 0, "")
183         self.qtgui_time_sink_x_1.enable_autoscale(True)
```

```
184         self.qtgui_time_sink_x_1.enable_grid(True)
185         self.qtgui_time_sink_x_1.enable_axis_labels(True)
186         self.qtgui_time_sink_x_1.enable_control_panel(True)
187         self.qtgui_time_sink_x_1.enable_stem_plot(False)
188 
189 
190         labels = ['Signal 1', 'Signal 2', 'Signal 3', 'Signal 4', '...
    Signal 5',
191             'Signal 6', 'Signal 7', 'Signal 8', 'Signal 9', 'Signal 10'...
    ]
192         widths = [1, 1, 1, 1, 1,
193             1, 1, 1, 1, 1]
194         colors = ['blue', 'red', 'green', 'black', 'cyan',
195             'magenta', 'yellow', 'dark red', 'dark green', 'dark blue']
196         alphas = [1.0, 1.0, 1.0, 1.0, 1.0,
197             1.0, 1.0, 1.0, 1.0, 1.0]
198         styles = [1, 1, 1, 1, 1,
199             1, 1, 1, 1, 1]
200         markers = [-1, -1, -1, -1, -1,
201             -1, -1, -1, -1, -1]
202 
203 
204         for i in range(1):
205             if len(labels[i]) == 0:
206                 self.qtgui_time_sink_x_1.set_line_label(i, "Data {0}"....
    format(i))
207             else:
208                 self.qtgui_time_sink_x_1.set_line_label(i, labels[i])
209             self.qtgui_time_sink_x_1.set_line_width(i, widths[i])
210             self.qtgui_time_sink_x_1.set_line_color(i, colors[i])
211             self.qtgui_time_sink_x_1.set_line_style(i, styles[i])
212             self.qtgui_time_sink_x_1.set_line_marker(i, markers[i])
213             self.qtgui_time_sink_x_1.set_line_alpha(i, alphas[i])
214 
215         self._qtgui_time_sink_x_1_win = sip.wrapinstance(self....
    qtgui_time_sink_x_1.qwidget(), Qt.QWidget)
216         self.top_layout.addWidget(self._qtgui_time_sink_x_1_win)
217         self.qtgui_time_sink_x_0_1 = qtgui.time_sink_c(
218             1024, #size
219             samp_rate, #samp_rate
220             'CSI output', #name
221             1, #number of inputs
222             None # parent
223         )
224         self.qtgui_time_sink_x_0_1.set_update_time(0.10)
225         self.qtgui_time_sink_x_0_1.set_y_axis(-1, 1)
226 
227         self.qtgui_time_sink_x_0_1.set_y_label('Amplitude', "")
228 
```

```
229         self.qtgui_time_sink_x_0_1.enable_tags(True)
230         self.qtgui_time_sink_x_0_1.set_trigger_mode(qtgui....
    TRIG_MODE_FREE, qtgui.TRIG_SLOPE_POS, 0.0, 0, 0, "")
231         self.qtgui_time_sink_x_0_1.enable_autoscale(True)
232         self.qtgui_time_sink_x_0_1.enable_grid(True)
233         self.qtgui_time_sink_x_0_1.enable_axis_labels(True)
234         self.qtgui_time_sink_x_0_1.enable_control_panel(True)
235         self.qtgui_time_sink_x_0_1.enable_stem_plot(False)
236 
237 
238         labels = ['', '', '', '', '',
239             '', '', '', '', '']
240         widths = [1, 1, 1, 1, 1,
241             1, 1, 1, 1, 1]
242         colors = ['red', 'red', 'green', 'black', 'cyan',
243             'magenta', 'yellow', 'dark red', 'dark green', 'dark blue']
244         alphas = [1.0, 1.0, 1.0, 1.0, 1.0,
245             1.0, 1.0, 1.0, 1.0, 1.0]
246         styles = [1, 1, 1, 1, 1,
247             1, 1, 1, 1, 1]
248         markers = [-1, -1, -1, -1, -1,
249             -1, -1, -1, -1, -1]
250 
251 
252         for i in range(2):
253             if len(labels[i]) == 0:
254                 if (i % 2 == 0):
255                     self.qtgui_time_sink_x_0_1.set_line_label(i, "Re{{...
    Data {0}}}".format(i/2))
256                 else:
257                     self.qtgui_time_sink_x_0_1.set_line_label(i, "Im{{...
    Data {0}}}".format(i/2))
258             else:
259                 self.qtgui_time_sink_x_0_1.set_line_label(i, labels[i])
260             self.qtgui_time_sink_x_0_1.set_line_width(i, widths[i])
261             self.qtgui_time_sink_x_0_1.set_line_color(i, colors[i])
262             self.qtgui_time_sink_x_0_1.set_line_style(i, styles[i])
263             self.qtgui_time_sink_x_0_1.set_line_marker(i, markers[i])
264             self.qtgui_time_sink_x_0_1.set_line_alpha(i, alphas[i])
265 
266         self._qtgui_time_sink_x_0_1_win = sip.wrapinstance(self....
    qtgui_time_sink_x_0_1.qwidget(), Qt.QWidget)
267         self.top_layout.addWidget(self._qtgui_time_sink_x_0_1_win)
268         self.qtgui_time_sink_x_0_0 = qtgui.time_sink_c(
269             1024, #size
270             samp_rate, #samp_rate
271             'Receiver Output', #name
272             1, #number of inputs
273             None # parent
```

```
274         )
275         self.qtgui_time_sink_x_0_0.set_update_time(0.10)
276         self.qtgui_time_sink_x_0_0.set_y_axis(-1, 1)
277 
278         self.qtgui_time_sink_x_0_0.set_y_label('Amplitude', "")
279 
280         self.qtgui_time_sink_x_0_0.enable_tags(True)
281         self.qtgui_time_sink_x_0_0.set_trigger_mode(qtgui....
    TRIG_MODE_FREE, qtgui.TRIG_SLOPE_POS, 0.0, 0, 0, "")
282         self.qtgui_time_sink_x_0_0.enable_autoscale(True)
283         self.qtgui_time_sink_x_0_0.enable_grid(True)
284         self.qtgui_time_sink_x_0_0.enable_axis_labels(True)
285         self.qtgui_time_sink_x_0_0.enable_control_panel(True)
286         self.qtgui_time_sink_x_0_0.enable_stem_plot(False)
287 
288 
289         labels = ['', '', '', '', '',
290             '', '', '', '', '']
291         widths = [1, 1, 1, 1, 1,
292             1, 1, 1, 1, 1]
293         colors = ['magenta', 'red', 'green', 'black', 'cyan',
294             'magenta', 'yellow', 'dark red', 'dark green', 'dark blue']
295         alphas = [1.0, 1.0, 1.0, 1.0, 1.0,
296             1.0, 1.0, 1.0, 1.0, 1.0]
297         styles = [1, 1, 1, 1, 1,
298             1, 1, 1, 1, 1]
299         markers = [-1, -1, -1, -1, -1,
300             -1, -1, -1, -1, -1]
301 
302 
303         for i in range(2):
304             if len(labels[i]) == 0:
305                 if (i % 2 == 0):
306                     self.qtgui_time_sink_x_0_0.set_line_label(i, "Re{{...
    Data {0}}}".format(i/2))
307                 else:
308                     self.qtgui_time_sink_x_0_0.set_line_label(i, "Im{{...
    Data {0}}}".format(i/2))
309             else:
310                 self.qtgui_time_sink_x_0_0.set_line_label(i, labels[i])
311             self.qtgui_time_sink_x_0_0.set_line_width(i, widths[i])
312             self.qtgui_time_sink_x_0_0.set_line_color(i, colors[i])
313             self.qtgui_time_sink_x_0_0.set_line_style(i, styles[i])
314             self.qtgui_time_sink_x_0_0.set_line_marker(i, markers[i])
315             self.qtgui_time_sink_x_0_0.set_line_alpha(i, alphas[i])
316 
317         self._qtgui_time_sink_x_0_0_win = sip.wrapinstance(self....
    qtgui_time_sink_x_0_0.qwidget(), Qt.QWidget)
318         self.top_layout.addWidget(self._qtgui_time_sink_x_0_0_win)
```

```
319         self.qtgui_time_sink_x_0 = qtgui.time_sink_f(
320             1024, #size
321             samp_rate, #samp_rate
322             'Scope Plot Output', #name
323             1, #number of inputs
324             None # parent
325         )
326         self.qtgui_time_sink_x_0.set_update_time(0.10)
327         self.qtgui_time_sink_x_0.set_y_axis(-1, 1)
328 
329         self.qtgui_time_sink_x_0.set_y_label('Amplitude', "")
330 
331         self.qtgui_time_sink_x_0.enable_tags(True)
332         self.qtgui_time_sink_x_0.set_trigger_mode(qtgui.TRIG_MODE_FREE,...
    qtgui.TRIG_SLOPE_POS, 0.0, 0, 0, "")
333         self.qtgui_time_sink_x_0.enable_autoscale(True)
334         self.qtgui_time_sink_x_0.enable_grid(True)
335         self.qtgui_time_sink_x_0.enable_axis_labels(True)
336         self.qtgui_time_sink_x_0.enable_control_panel(True)
337         self.qtgui_time_sink_x_0.enable_stem_plot(False)
338 
339 
340         labels = ['', '', '', '', '',
341             '', '', '', '', '']
342         widths = [1, 1, 1, 1, 1,
343             1, 1, 1, 1, 1]
344         colors = ['red', 'red', 'green', 'black', 'cyan',
345             'magenta', 'yellow', 'dark red', 'dark green', 'dark blue']
346         alphas = [1.0, 1.0, 1.0, 1.0, 1.0,
347             1.0, 1.0, 1.0, 1.0, 1.0]
348         styles = [1, 1, 1, 1, 1,
349             1, 1, 1, 1, 1]
350         markers = [-1, -1, -1, -1, -1,
351             -1, -1, -1, -1, -1]
352 
353 
354         for i in range(1):
355             if len(labels[i]) == 0:
356                 self.qtgui_time_sink_x_0.set_line_label(i, "Data {0}"....
   format(i))
357             else:
358                 self.qtgui_time_sink_x_0.set_line_label(i, labels[i])
359             self.qtgui_time_sink_x_0.set_line_width(i, widths[i])
360             self.qtgui_time_sink_x_0.set_line_color(i, colors[i])
361             self.qtgui_time_sink_x_0.set_line_style(i, styles[i])
362             self.qtgui_time_sink_x_0.set_line_marker(i, markers[i])
363             self.qtgui_time_sink_x_0.set_line_alpha(i, alphas[i])
364 
365         self._qtgui_time_sink_x_0_win = sip.wrapinstance(self....
```

```
    qtgui_time_sink_x_0.qwidget(), Qt.QWidget)
366         self.top_layout.addWidget(self._qtgui_time_sink_x_0_win)
367         self.qtgui_freq_sink_x_0_0 = qtgui.freq_sink_c(
368             1024, #size
369             window.WIN_BLACKMAN_hARRIS, #wintype
370             center_freq, #fc
371             band_width, #bw
372             "", #name
373             1,
374             None # parent
375         )
376         self.qtgui_freq_sink_x_0_0.set_update_time(0.10)
377         self.qtgui_freq_sink_x_0_0.set_y_axis((-140), 10)
378         self.qtgui_freq_sink_x_0_0.set_y_label('Relative Gain', 'dB')
379         self.qtgui_freq_sink_x_0_0.set_trigger_mode(qtgui....
    TRIG_MODE_FREE, 0.0, 0, "")
380         self.qtgui_freq_sink_x_0_0.enable_autoscale(True)
381         self.qtgui_freq_sink_x_0_0.enable_grid(True)
382         self.qtgui_freq_sink_x_0_0.set_fft_average(0.05)
383         self.qtgui_freq_sink_x_0_0.enable_axis_labels(True)
384         self.qtgui_freq_sink_x_0_0.enable_control_panel(True)
385         self.qtgui_freq_sink_x_0_0.set_fft_window_normalized(False)
386
387
388
389         labels = ['', '', '', '', '',
390             '', '', '', '', '']
391         widths = [1, 1, 1, 1, 1,
392             1, 1, 1, 1, 1]
393         colors = ["blue", "red", "green", "black", "cyan",
394             "magenta", "yellow", "dark red", "dark green", "dark blue"]
395         alphas = [1.0, 1.0, 1.0, 1.0, 1.0,
396             1.0, 1.0, 1.0, 1.0, 1.0]
397
398         for i in range(1):
399             if len(labels[i]) == 0:
400                 self.qtgui_freq_sink_x_0_0.set_line_label(i, "Data {0}"...
    .format(i))
401             else:
402                 self.qtgui_freq_sink_x_0_0.set_line_label(i, labels[i])
403             self.qtgui_freq_sink_x_0_0.set_line_width(i, widths[i])
404             self.qtgui_freq_sink_x_0_0.set_line_color(i, colors[i])
405             self.qtgui_freq_sink_x_0_0.set_line_alpha(i, alphas[i])
406
407         self._qtgui_freq_sink_x_0_0_win = sip.wrapinstance(self....
    qtgui_freq_sink_x_0_0.qwidget(), Qt.QWidget)
408         self.top_layout.addWidget(self._qtgui_freq_sink_x_0_0_win)
409         self._noise_voltage_range = qtgui.Range(0, 1, .01, 0.1, 200)
410         self._noise_voltage_win = qtgui.RangeWidget(self....
```

```
_noise_voltage_range, self.set_noise_voltage, "Noise Amplitude", "...
counter_slider", float, QtCore.Qt.Horizontal)
411         self.top_layout.addWidget(self._noise_voltage_win)
412         self._freq_offset_range = qtgui.Range(-3, 3, .01, 0, 200)
413         self._freq_offset_win = qtgui.RangeWidget(self....
_freq_offset_range, self.set_freq_offset, "Frequency Offset (...
Multiples of Sub-carrier spacing)", "counter_slider", float, QtCore....
Qt.Horizontal)
414         self.top_layout.addWidget(self._freq_offset_win)
415         self.epy_block_0 = epy_block_0.blk(fft_len=fft_len, ...
pilot_carriers=((-6,-5,5,6),), pilot_symbols=((-1,1,-1,1),))
416         self.digital_ofdm_rx_0 = digital.ofdm_rx(
417             fft_len=fft_len, cp_len=(fft_len//4),
418             frame_length_tag_key='frame_'+"rx_len",
419             packet_length_tag_key="rx_len",
420             occupied_carriers=((-4,-3,-2,-1,1,2,3,4),),
421             pilot_carriers=((-6,-5,5,6),),
422             pilot_symbols=((-1,1,-1,1),),
423             sync_word1=None,
424             sync_word2=None,
425             bps_header=1,
426             bps_payload=2,
427             debug_log=False,
428             scramble_bits=False)
429         self.blocks_uchar_to_float_0 = blocks.uchar_to_float()
430         self.blocks_tag_debug_0 = blocks.tag_debug(gr.sizeof_char*1, '...
Rx Packets', "")
431         self.blocks_tag_debug_0.set_display(False)
432         self.blocks_nlog10_ff_0 = blocks.nlog10_ff(10, 1, 0)
433         self.blocks_multiply_const_vxx_0 = blocks.multiply_const_ff(10)
434         self.blocks_moving_average_xx_0 = blocks.moving_average_ff...
(1024, 1, 4000, 1)
435         self.blocks_file_sink_2 = blocks.file_sink(gr.sizeof_gr_complex...
*1, '/home/acps/Documents/data/Velocity_track_CSI_data', False)
436         self.blocks_file_sink_2.set_unbuffered(False)
437         self.blocks_file_sink_1 = blocks.file_sink(gr.sizeof_float*1, '...
/home/acps/Documents/data/Velocity_Track_RSSI_data', False)
438         self.blocks_file_sink_1.set_unbuffered(False)
439         self.blocks_file_sink_0 = blocks.file_sink(gr.sizeof_float*1, '...
/home/acps/Documents/data/velocity_track_OFDM_rx', False)
440         self.blocks_file_sink_0.set_unbuffered(False)
441         self.blocks_complex_to_mag_squared_0 = blocks....
complex_to_mag_squared(1)
442         self.blocks_add_const_vxx_1 = blocks.add_const_ff(10)
443
444
445         ##################################################
446         # Connections
447         ##################################################
```

```
448         self.connect((self.blocks_add_const_vxx_1, 0), (self....
    blocks_multiply_const_vxx_0, 0))
449         self.connect((self.blocks_complex_to_mag_squared_0, 0), (self....
    blocks_moving_average_xx_0, 0))
450         self.connect((self.blocks_moving_average_xx_0, 0), (self....
    blocks_nlog10_ff_0, 0))
451         self.connect((self.blocks_multiply_const_vxx_0, 0), (self....
    blocks_file_sink_1, 0))
452         self.connect((self.blocks_multiply_const_vxx_0, 0), (self....
    qtgui_time_sink_x_1, 0))
453         self.connect((self.blocks_multiply_const_vxx_0, 0), (self....
    qtgui_waterfall_sink_x_0, 0))
454         self.connect((self.blocks_nlog10_ff_0, 0), (self....
    blocks_add_const_vxx_1, 0))
455         self.connect((self.blocks_uchar_to_float_0, 0), (self....
    blocks_file_sink_0, 0))
456         self.connect((self.blocks_uchar_to_float_0, 0), (self....
    qtgui_time_sink_x_0, 0))
457         self.connect((self.digital_ofdm_rx_0, 0), (self....
    blocks_tag_debug_0, 0))
458         self.connect((self.digital_ofdm_rx_0, 0), (self....
    blocks_uchar_to_float_0, 0))
459         self.connect((self.epy_block_0, 0), (self.blocks_file_sink_2, ...
    0))
460         self.connect((self.epy_block_0, 0), (self.qtgui_time_sink_x_0_1...
    , 0))
461         self.connect((self.epy_block_0, 0), (self....
    qtgui_waterfall_sink_x_0_0, 0))
462         self.connect((self.uhd_usrp_source_0, 0), (self....
    blocks_complex_to_mag_squared_0, 0))
463         self.connect((self.uhd_usrp_source_0, 0), (self....
    digital_ofdm_rx_0, 0))
464         self.connect((self.uhd_usrp_source_0, 0), (self.epy_block_0, 0)...
    )
465         self.connect((self.uhd_usrp_source_0, 0), (self....
    qtgui_freq_sink_x_0_0, 0))
466         self.connect((self.uhd_usrp_source_0, 0), (self....
    qtgui_time_sink_x_0_0, 0))
467 
468 
469     def closeEvent(self, event):
470         self.settings = Qt.QSettings("GNU Radio", "ofdm_Rx")
471         self.settings.setValue("geometry", self.saveGeometry())
472         self.stop()
473         self.wait()
474 
475         event.accept()
476 
477     def get_samp_rate(self):
```

```
478             return self.samp_rate
479
480         def set_samp_rate(self, samp_rate):
481             self.samp_rate = samp_rate
482             self.qtgui_time_sink_x_0.set_samp_rate(self.samp_rate)
483             self.qtgui_time_sink_x_0_0.set_samp_rate(self.samp_rate)
484             self.qtgui_time_sink_x_0_1.set_samp_rate(self.samp_rate)
485             self.qtgui_time_sink_x_1.set_samp_rate(self.samp_rate)
486             self.uhd_usrp_source_0.set_samp_rate(self.samp_rate)
487
488         def get_packet_length_tag_key(self):
489             return self.packet_length_tag_key
490
491         def set_packet_length_tag_key(self, packet_length_tag_key):
492             self.packet_length_tag_key = packet_length_tag_key
493
494         def get_packet_len(self):
495             return self.packet_len
496
497         def set_packet_len(self, packet_len):
498             self.packet_len = packet_len
499
500         def get_noise_voltage(self):
501             return self.noise_voltage
502
503         def set_noise_voltage(self, noise_voltage):
504             self.noise_voltage = noise_voltage
505
506         def get_len_tag_key(self):
507             return self.len_tag_key
508
509         def set_len_tag_key(self, len_tag_key):
510             self.len_tag_key = len_tag_key
511
512         def get_freq_offset(self):
513             return self.freq_offset
514
515         def set_freq_offset(self, freq_offset):
516             self.freq_offset = freq_offset
517
518         def get_fft_len(self):
519             return self.fft_len
520
521         def set_fft_len(self, fft_len):
522             self.fft_len = fft_len
523             self.epy_block_0.fft_len = self.fft_len
524
525         def get_center_freq(self):
526             return self.center_freq
```

```
527
528     def set_center_freq(self, center_freq):
529         self.center_freq = center_freq
530         self.qtgui_freq_sink_x_0_0.set_frequency_range(self.center_freq...
    , self.band_width)
531         self.qtgui_waterfall_sink_x_0.set_frequency_range(self....
    center_freq, self.band_width)
532         self.qtgui_waterfall_sink_x_0_0.set_frequency_range(self....
    center_freq, self.band_width)
533         self.uhd_usrp_source_0.set_center_freq(self.center_freq, 0)
534
535     def get_band_width(self):
536         return self.band_width
537
538     def set_band_width(self, band_width):
539         self.band_width = band_width
540         self.qtgui_freq_sink_x_0_0.set_frequency_range(self.center_freq...
    , self.band_width)
541         self.qtgui_waterfall_sink_x_0.set_frequency_range(self....
    center_freq, self.band_width)
542         self.qtgui_waterfall_sink_x_0_0.set_frequency_range(self....
    center_freq, self.band_width)
543
544
545
546
547 def main(top_block_cls=ofdm_Rx, options=None):
548
549     qapp = Qt.QApplication(sys.argv)
550
551     tb = top_block_cls()
552
553     tb.start()
554
555     tb.show()
556
557     def sig_handler(sig=None, frame=None):
558         tb.stop()
559         tb.wait()
560
561         Qt.QApplication.quit()
562
563     signal.signal(signal.SIGINT, sig_handler)
564     signal.signal(signal.SIGTERM, sig_handler)
565
566     timer = Qt.QTimer()
567     timer.start(500)
568     timer.timeout.connect(lambda: None)
569
```

```
570     qapp.exec_()
571
572 if __name__ == '__main__':
573     main()
```

Listing A.2: OFDM Receiver

A.2 Custom CSI Estimator Block

```
1  # Code implemented to calculate Channel State Information (CSI)
2
3  import numpy as np
4  from gnuradio import gr
5
6  class blk(gr.basic_block):
7      def __init__(self):
8          gr.basic_block.__init__(self,
9              name="Custom Block",
10             in_sig=[np.complex64],
11             out_sig=[np.complex64])
12
13     def general_work(self, input_items, output_items):
14         in0 = input_items[0]
15         out = output_items[0]
16         # Implement custom processing here
17         out[:] = in0 * 2  # Example: amplify signal by 2
18         self.consume(0, len(in0))
19         return len(out)
```

Listing A.3: CSI Estimation

Appendix B

Drone

B.1 Drone Count

```
from codrone_edu.drone import *
import threading
import time
from pathlib import Path

Drones = []
n_drones = 5

def get_ports():
    ports = []

    for i in range(n_drones):
        file = "/dev/ttyACM"
        file_path = Path(f"{file}{i}")
        if file_path.exists() == True:
            ports.append(f"{file}{i}")

    return ports

def drone_path(path:str):
    drone = Drone()
    drone.pair(path)
    drone.takeoff()
    #drone.circle()
    for i in range(3):
        drone.sway(50,1)
    drone.land()
    drone.close()

if __name__ == "__main__":
    ports = get_ports()
```

```
32      print(ports)
33 
34      for i in range(len(ports)):
35          drone = threading.Thread(target=drone_path, args=(ports[i],))
36          Drones.append(drone)
37          drone.start()
38 
39      # Join the threads
40      for drone in Drones:
41          drone.join()
```

Listing B.1: Code used for collecting drone count dataset

B.2 UAV Velocity

```
1  from codrone_edu.drone import *
2  import threading
3  import time
4  from pathlib import Path
5  import random
6  import logging
7 
8  logging.basicConfig(filename='drone_log.txt', level=logging.INFO, ...
       format='%(asctime)s - %(levelname)s - %(message)s')
9 
10 Drones = []
11 n_drones = 5
12 speed = [0.1, 0.6, 1, 1.5, 2]
13 
14 def get_ports():
15     ports = []
16 
17     for i in range(n_drones):
18         file = "/dev/ttyACM"
19         file_path = Path(f"{file}{i}")
20         if file_path.exists():
21             ports.append(f"{file}{i}")
22 
23     return ports
24 
25 def drone_path(path:str):
26     drone = Drone()
27     drone.pair(path)
28     drone.takeoff()
29     time.sleep(5)
30 
31     overall_start_time = time.time()
32     speed_list = speed.copy()
```

```
33      random.shuffle(speed_list)
34      current_speed = speed_list.pop()
35      speed_change_interval = 30  # seconds
36      next_speed_change_time = time.time() + speed_change_interval
37      total_runtime = 3 * 60  # 5 minutes in seconds
38
39      while time.time() - overall_start_time < total_runtime:
40          if time.time() > next_speed_change_time:
41              if not speed_list:
42                  speed_list = speed.copy()
43                  random.shuffle(speed_list)
44              current_speed = speed_list.pop()
45              next_speed_change_time = time.time() + ...
    speed_change_interval
46
47          duration = max(1, 1 * (1 / current_speed))
48
49          angular_speed_before_forward = drone.get_x_accel()
50          logging.info(f"Moving forward - Angular Speed: {...
    angular_speed_before_forward}, Current Speed: {current_speed}, ...
    Duration: {duration}")
51          drone.move_forward(distance=150, units="cm", speed=...
    current_speed)
52          time.sleep(duration)
53
54          angular_speed_after_forward = drone.get_x_accel()
55          logging.info(f"Completed forward - Angular Speed: {...
    angular_speed_after_forward}, Current Speed: {current_speed}, ...
    Duration: {duration}")
56
57          angular_speed_before_backward = drone.get_x_accel()
58          logging.info(f"Moving backward - Angular Speed: {...
    angular_speed_before_backward}, Current Speed: {current_speed}, ...
    Duration: {duration}")
59          drone.move_backward(distance=150, units="cm", speed=...
    current_speed)
60          time.sleep(duration)
61          angular_speed_after_backward = drone.get_x_accel()
62          logging.info(f"Completed backward - Angular Speed: {...
    angular_speed_after_backward}, Current Speed: {current_speed}, ...
    Duration: {duration}")
63
64      drone.land()
65      drone.close()
66
67  if __name__ == "__main__":
68      ports = get_ports()
69      print(ports)
70      if ports:
```

```
71        drone_path(ports[0])
```

Listing B.2: Code for collecting the data for UAV velocity

Appendix C

CNN

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3  from pathlib import Path
4  import os
5  from sklearn.model_selection import train_test_split
6  import tensorflow as tf
7  from tensorflow.keras import layers, models, optimizers, callbacks, ...
       regularizers
8
9  class DataLoader:
10     def __init__(self, base_path, samples_to_take=1_000_000, ...
       sequence_length=256):
11         self.base_path = base_path
12         self.samples_to_take = samples_to_take
13         self.sequence_length = sequence_length
14         self.data_list = []
15         self.labels_list = []
16
17     def load_data(self):
18         for i in range(1, 5):
19             file_path = f"{self.base_path}{i}"
20             label = i - 1
21             file_data = np.fromfile(open(file_path, 'rb'), dtype=np....
       complex64)
22             start_idx = len(file_data) // 2 - self.samples_to_take // 2
23             end_idx = start_idx + self.samples_to_take
24             real_part = self.normalize(file_data.real[start_idx:end_idx...
       ])
25             imag_part = self.normalize(file_data.imag[start_idx:end_idx...
       ])
26             abs_part = self.normalize(np.abs(file_data)[start_idx:...
       end_idx])
27             combined_data = np.column_stack((real_part, imag_part, ...
       abs_part))
```

```
28              self.data_list.append(combined_data)
29              self.labels_list.extend([label] * len(combined_data))
30          self.data = np.vstack(self.data_list)
31          self.labels = np.array(self.labels_list)
32          self.reshape_data()
33
34      def normalize(self, data):
35          return (data - np.mean(data)) / np.std(data)
36
37      def reshape_data(self):
38          num_samples = self.data.shape[0] // self.sequence_length
39          self.data = self.data[:num_samples * self.sequence_length]
40          self.labels = self.labels[:num_samples * self.sequence_length]
41          self.data = self.data.reshape(num_samples, self.sequence_length...
    , 3)
42          self.labels = self.labels[::self.sequence_length]
43
44      def get_train_test_split(self, test_size=0.2, random_state=42):
45          return train_test_split(self.data, self.labels, test_size=...
    test_size, random_state=random_state)
46
47  class ResNetModel:
48      def __init__(self, input_shape, channels, num_classes=7):
49          self.input_shape = input_shape
50          self.channels = channels
51          self.num_classes = num_classes
52          self.model = self.build_model()
53
54      def residual_block(self, x, filters, kernel_size=3, stride=1, ...
    conv_shortcut=False):
55          shortcut = x
56          if conv_shortcut:
57              shortcut = layers.Conv1D(filters, 1, strides=stride, ...
    kernel_regularizer=regularizers.l2(0.001))(shortcut)
58              shortcut = layers.BatchNormalization()(shortcut)
59          x = layers.Conv1D(filters, kernel_size, strides=stride, padding...
    ='same', kernel_regularizer=regularizers.l2(0.001))(x)
60          x = layers.BatchNormalization()(x)
61          x = layers.ReLU()(x)
62          x = layers.Conv1D(filters, kernel_size, padding='same', ...
    kernel_regularizer=regularizers.l2(0.001))(x)
63          x = layers.BatchNormalization()(x)
64          x = layers.add([shortcut, x])
65          x = layers.ReLU()(x)
66          return x
67
68      def build_model(self):
69          input_layers = []
70          outputs = []
```

```
        for _ in range(self.channels):
            inputs = layers.Input(shape=self.input_shape)
            x = layers.Conv1D(8, 7, strides=2, padding='same', ...
kernel_regularizer=regularizers.l2(0.001))(inputs)
            x = layers.BatchNormalization()(x)
            x = layers.ReLU()(x)
            x = layers.MaxPooling1D(3, strides=2, padding='same')(x)

            x = self.residual_block(x, 8)
            x = self.residual_block(x, 16, stride=2, conv_shortcut=True...
)
            # x = self.residual_block(x, 16)
            # x = self.residual_block(x, 32, stride=2, conv_shortcut=...
True)

            x = layers.GlobalAveragePooling1D()(x)
            x = layers.Dropout(0.5)(x)
            input_layers.append(inputs)
            outputs.append(x)

        merged = layers.concatenate(outputs)
        output = layers.Dense(self.num_classes, activation='softmax')(...
merged)
        model = models.Model(inputs=input_layers, outputs=output)
        return model

    def compile_model(self, learning_rate=0.0001):
        optimizer = optimizers.Adam(learning_rate=learning_rate)
        self.model.compile(optimizer=optimizer, loss='...
sparse_categorical_crossentropy', metrics=['accuracy'])
        self.model.summary()
        tf.keras.utils.plot_model(self.model)

    def train_model(self, X_train, y_train, X_test, y_test, batch_size...
=64, epochs=50):
        class CustomCallback(callbacks.Callback):
            def on_epoch_end(self, epoch, logs=None):
                self.best_val_loss = min(self.best_val_loss, logs['...
val_loss'])

            def on_train_begin(self, logs=None):
                self.best_val_loss = float('inf')

        custom_callback = CustomCallback()

        callbacks_list = [
            custom_callback,
```

```
112             callbacks.EarlyStopping(monitor='val_loss', patience=10, ...
    restore_best_weights=True),
113             callbacks.ReduceLROnPlateau(monitor='val_loss', factor=0.5,...
     patience=5, min_lr=1e-7)
114         ]
115 
116         history = self.model.fit(
117             X_train, y_train,
118             validation_data=(X_test, y_test),
119             epochs=epochs,
120             batch_size=batch_size,
121             callbacks=callbacks_list,
122             verbose=1
123         )
124 
125         best_val_loss = custom_callback.best_val_loss
126         return history, best_val_loss
127 
128     def save_model(self, file_path):
129         self.model.save(file_path)
130 
131 class Plotter:
132     @staticmethod
133     def plot_history(history, best_val_loss, accuracy_path, loss_path):
134         plt.figure(figsize=(12, 4))
135         plt.subplot(1, 2, 1)
136         plt.plot(history.history['accuracy'], color='blue', linestyle='...
    -', marker='x')
137         plt.plot(history.history['val_accuracy'], color='orange', ...
    linestyle='-', marker='o')
138         plt.title(f'Model Accuracy - Best Val Loss: {best_val_loss:.4f}...
    ')
139         plt.ylabel('Accuracy')
140         plt.xlabel('Epoch')
141         plt.legend(['Train', 'Validation'], loc='upper left')
142         plt.savefig(accuracy_path)
143         plt.subplot(1, 2, 2)
144         plt.plot(history.history['loss'], color='blue', linestyle='-', ...
    marker='x')
145         plt.plot(history.history['val_loss'], color='orange', linestyle...
    ='-', marker='o')
146         plt.title(f'Model Loss - Best Val Loss: {best_val_loss:.4f}')
147         plt.ylabel('Loss')
148         plt.xlabel('Epoch')
149         plt.legend(['Train', 'Validation'], loc='upper left')
150         plt.savefig(loss_path)
151         plt.show()
152 
153 def main():
```

```
154     # Data loading and preprocessing
155     base_path = "../data/Drone_Count_CSI_data_Big"
156     data_loader = DataLoader(base_path)
157     data_loader.load_data()
158     X_train, X_test, y_train, y_test = data_loader.get_train_test_split...
    ()
159 
160     # Separate the data into different features
161     sequence_length = 256
162     X_train_real = X_train[:, :, 0].reshape(-1, sequence_length, 1)
163     X_train_imag = X_train[:, :, 1].reshape(-1, sequence_length, 1)
164     X_train_abs = X_train[:, :, 2].reshape(-1, sequence_length, 1)
165 
166     X_test_real = X_test[:, :, 0].reshape(-1, sequence_length, 1)
167     X_test_imag = X_test[:, :, 1].reshape(-1, sequence_length, 1)
168     X_test_abs = X_test[:, :, 2].reshape(-1, sequence_length, 1)
169 
170     X_train_inputs = [X_train_real, X_train_imag, X_train_abs]
171     X_test_inputs = [X_test_real, X_test_imag, X_test_abs]
172 
173     # Build, compile and train the model
174     input_shape = (sequence_length, 1)
175     channels = len(X_train_inputs)
176     resnet_model = ResNetModel(input_shape, channels)
177     resnet_model.compile_model()
178 
179     history, best_val_loss = resnet_model.train_model(X_train_inputs, ...
    y_train, X_test_inputs, y_test)
180 
181     # Save the model
182     model_file_path = f'../models/resnet_model_val_loss_{best_val_loss...
    :.4f}.keras'
183     resnet_model.save_model(model_file_path)
184 
185     # Plot the training history
186     accuracy_plot_path = f'../Img/resnet_accuracy_val_loss_{...
    best_val_loss:.4f}.png'
187     loss_plot_path = f'../Img/resnet_loss_val_loss_{best_val_loss:.4f}....
    png'
188     Plotter.plot_history(history, best_val_loss, accuracy_plot_path, ...
    loss_plot_path)
189 
190 if __name__ == "__main__":
191     main()
```

Listing C.1: CNN model code