WIRELESS COMMUNICATION SIGNAL GENERATION FOR DRONE DETECTION MODELS IN UNREAL ENGINE

by

Yusuf Özben

B.S., Electronics and Communication Engineering, Istanbul Technical University, 2020

Submitted to the Institute for Graduate Studies in
Science and Engineering in partial fulfillment of
the requirements for the degree of
Master of Science

Graduate Program in Computer Engineering
Boğaziçi University

2024

ACKNOWLEDGEMENTS

I would like to express my gratitude to my thesis advisor, Hüseyin Birkan Yılmaz, and all my professors for their unwavering support and contributions during these challenging times.

I'm deeply thankful to my research team and friends at NetLab, Boğaziçi University's Computer Engineering Department, with special appreciation for Süleyman Emre Demir, Furkan Oruç, and Kerem Avcı for their invaluable contributions.

I am grateful to my family, friends, and my girlfriend, Rojda, for their unwavering support.

This thesis is supported by the Boğaziçi University BAP under the grant number BAP-SUP-17862.

ABSTRACT

WIRELESS COMMUNICATION SIGNAL GENERATION FOR DRONE DETECTION MODELS IN UNREAL ENGINE

Drones have gained widespread popularity with different uses, including positive and malicious applications. Consequently, safeguarding territories from unauthorized drone activities has become more critical. However, the creation and testing of drone detection systems inherits challenges for some researchers due to the prohibitive costs of required equipment. In this thesis, we introduce an innovative approach by developing an Airsim GUI tool compatible with the Unreal Engine. This tool enables the generation of image and radio frequency (RF) datasets for drone detection within a simulation environment on a single screen. Our novel approach involves modeling the USRP receiver to collect RF signals as raw in-phase and quadrature (IQ) data. In addition, users of this tool can manage automated route planning, recording configurations, various cameras, and RF configurations. Researchers can now create datasets with diverse images and RF configurations without physical drones, cameras, or USRPs, facilitating experimentation with various drone detection models. Additionally, we demonstrate the use of the implemented tool across different use cases and propose drone detection and distance estimation models.

ÖZET

UNREAL MOTORUNDA İHA ALGILAMA MODELLERİ İÇİN KABLOSUZ İLETİŞİM SİNYALİ ÜRETİMİ

İnsansız hava araçları (İHA'lar), farklı kullanım alanlarıyla geniş bir popülerlik kazanmıştır, bu alanlar arasında olumlu ve kötü niyetli uygulamalar bulunmaktadır. Bu nedenle, bölgeleri yetkisiz İHA faaliyetlerinden korumak daha da önemli hale gelmiştir. Ancak, İHA tespit sistemlerinin oluşturulması ve test edilmesi, bazı araştırmacılar için gerekli ekipmanların yüksek maliyetleri nedeniyle zorluklar içermektedir. Bu tezde, Unreal Engine ile uyumlu bir Airsim GUI aracı geliştirerek bu soruna yenilikçi bir yaklaşım sunuyoruz. Bu araç, tek bir ekran üzerinde simülasyon ortamında İHA tespiti için görüntü ve radyo frekansı (RF) veri setlerinin oluşturulmasını sağlar. Yenilikçi yaklaşımımız, USRP alıcısını modelleyerek RF sinyallerini ham faz içi ve dik evreli (IQ) verileri olarak toplamayı içerir. Ayrıca, bu aracın kullanıcıları otomatik rota planlamayı, kayıt konfigürasyonlarını, çeşitli kamera ve RF konfigürasyonlarını yönetebilirler. Araştırmacılar artık fiziksel İHA'lar, kameralar veya USRP'ler olmadan çeşitli İHA tespit modelleri üzerinde deneme yapmayı kolaylaştıran çeşitli görüntü ve RF konfigürasyonlarına sahip veri setleri oluşturabilirler. Ayrıca, geliştirilen aracın farklı kullanım senaryolarında kullanımını gösteriyor ve İHA tespiti ile mesafe tahmini modelleri öneriyoruz.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS . . . iii
ABSTRACT . . . iv
ÖZET . . . v
LIST OF FIGURES . . . ix
LIST OF TABLES . . . xii
LIST OF SYMBOLS . . . xiii
LIST OF ACRONYMS/ABBREVIATIONS . . . xv
1. INTRODUCTION . . . 1
1.1. Problem Definition . . . 2
1.2. Key Contributions . . . 2
1.3. Thesis Outline . . . 2
2. RELATED WORKS . . . 4
2.1. Drone Detection Datasets . . . 4
2.1.1. Image Datasets . . . 4
2.1.2. RF Datasets . . . 5
2.2. Drone Detection by Using RF Data . . . 6
2.3. Drone Detection by Using a Camera . . . 7
2.4. Drone Detection in Simulation Environments . . . 7
3. SIMULATION ENVIRONMENT SETUP FOR DRONE DETECTION . . . 9
3.1. Simulation Architecture . . . 9
3.2. Used Technologies . . . 10
3.2.1. Unreal Engine . . . 10
3.2.2. Airsim . . . 11
3.2.3. Pyphysim . . . 12
3.2.4. PyQt . . . 12
3.3. Unreal Engine Level Design . . . 13
3.4. Airsim GUI Development . . . 16
3.4.1. Airsim GUI . . . 16

3.4.1.1. Connect to Server 16
3.4.1.2. Tab Widget 18
3.4.1.3. Logging 19
3.4.2. Airsim GUI Recording Structure 19
3.4.3. Airsim GUI RF Data Generation 20
3.5. Airsim GUI Workflow 22
3.5.1. Unreal Engine Environment Setup 23
3.5.2. Airsim Configuration 23
3.5.3. Preparing the Table of the Detection System 26
3.5.4. Running the Unreal Engine Level 27
3.5.5. Connecting to the Server 28
3.5.6. Adding Additional Drones 28
3.5.7. Setting the RF Parameters 28
3.5.8. Setting the Drone Path 32
3.5.9. Recording the Experiment 33
3.5.10. Offline Data Generation 34
4. RF DATA MODEL AND GENERATION 36
4.1. Pyphysim Library Usage 36
4.2. RF Generation and Channel Model 36
4.2.1. Airsim GUI RF Parameters 36
4.2.2. RF Generation 37
4.2.3. RF Data Generation Steps 37
4.2.3.1. Calculate the Number of Symbols 39
4.2.3.2. Markov Chain 40
4.2.3.3. RNG for Symbol Generation 41
4.2.3.4. Modulation 41
4.2.3.5. OFDM (Digital) Modulation 42
4.2.3.6. Transmitter Power and Transmitter Gain 43
4.2.3.7. Jakes' Channel Model 44
4.2.3.8. TDL Channel Profile 45
4.2.3.9. Single User Channel 45

4.2.3.10. Corrupt Data 45
4.2.3.11. Path Loss 46
4.2.3.12. Phase Shift 46
4.2.3.13. Sum Other Drones' RF Signals 46
4.2.3.14. Additive White Gaussian Noise 47
4.2.3.15. Receiver Gain 47
4.2.3.16. USRP 47
5. EXAMPLE USE CASES 48
5.1. Drone Detection via Image Data 49
5.2. Drone Detection via RF Data 53
5.3. Drone Distance Detection via RF Data 55
6. CONTRIBUTIONS AND FUTURE WORK 58
6.1. Summary of Contributions 58
6.2. Future Work 58
REFERENCES 59

LIST OF FIGURES

Figure 3.1. The simulation component architecture. 9

Figure 3.2. Blueprint example: adding actions to keyboard commands for adding birds to the simulation environment. 11

Figure 3.3. PyQt designer. 13

Figure 3.4. Map image of the selected simulation environment. 14

Figure 3.5. Aptullah Kuran Library. 14

Figure 3.6. Screenshot images from the simulation environment. 15

Figure 3.7. Table settings. 17

Figure 3.8. Located cameras based on configured table settings. 18

Figure 3.9. Example of generated vehicle and image dataset folder hierarchy. . 21

Figure 3.10. The USRP coverage in our simulation system shows that drone controllers are always out of coverage. 22

Figure 3.11. High-level RF data generation steps. 22

Figure 3.12. Example of generated RF dataset folder hierarchy. 22

Figure 3.13. Airsim “SimMode” and “ViewMode” settings. 23

Figure 3.14. Airsim “Recording” setting. 24

Figure 3.15. Airsim “SubWindows” settings. 24

Figure 3.16. Unreal Engine with Airsim. Three small images at the bottom of the figure are subwindows in Airsim. 25

Figure 3.17. Camera types and default camera settings in Airsim. 26

Figure 3.18. Airsim “Vehicles” and “ExternalCameras” settings. 27

Figure 3.19. Airsim GUI “Guide” tab. 28

Figure 3.20. Airsim GUI “Drones” tab. 29

Figure 3.21. Airsim GUI “RF Parameters” tab. 29

Figure 3.22. An example of TDL channel settings. 30

Figure 3.23. An example of USRP settings. 30

Figure 3.24. An example of Markov chain probability and metadata settings. . 31

Figure 3.25. Airsim GUI “Set Drone Path” tab. 32

Figure 3.26. Route planning example in the Airsim GUI. 32

Figure 3.27. Airsim GUI “Recording” tab. 33

Figure 3.28. Segmented drone (ground truth) image example. 34

Figure 3.29. Airsim GUI “Offline Data Generation” tab. 35

Figure 4.1. RF generation detailed diagram. 38

Figure 4.2. An example of Markov chain. 40

Figure 4.3. QPSK modulated data in IQ plane. 41

Figure 4.4. OFDM Diagram. 42

Figure 4.5. Corrupt data algorithm. 46

Figure 5.1. Dataset scenarios. 48

Figure 5.2. YOLOv8 network architecture. 51

Figure 5.3. ROC curves for drone detection with two cameras and a single camera. 53

Figure 5.4. Input matrix format of drone distance detection model. 56

Figure 5.5. Drone distance detection CNN model. 56

LIST OF TABLES

Table 1.1. Advantages and disadvantages drone detection methods. 3

Table 5.1. YOLOv8 object detection model performance comparison. 49

Table 5.2. YOLOv8n training process for drone detection via image dataset. . 52

Table 5.3. Confusion matrices of drone detection by using a single and two cameras. 54

Table 5.4. YOLOv8 classification model performance comparison. 55

Table 5.5. YOLOv8n-cls classification training process for drone detection via RF dataset. 55

Table 5.6. Drone distance detection statistics for the absolute difference and percentage error between ground truth and estimations. 57

LIST OF SYMBOLS

A_{Tx}	Transmitter amplitude
c	Speed of light
d	distance
f_c	Carrier frequency
F_d	Doppler frequency
$G_{\mathrm{Tx}}(dB)$	Transmitter gain in dB
G_{Tx}	Transmitter gain in linear scale
$h(t)$	Channel matrix
L	Number of rays in Jakes' model
L_{ofdm}	Output symbol length of OFDM block
L_{oi}	Input symbol length of OFDM block
L_{rb}	RF block length of OFDM
L_{sl}	Required symbol count between two consecutive logs
L_{ss}	Symbol length of a Markov state
L_{zp}	Required zero padding for input of OFDM block
n_{cp}	CP size of OFDM
n_{di}	Delay index for TDL channel
n_{fs}	FFT size of OFDM
n_{iqs}	Number of IQ signal per second parameter
n_{rb}	Generated RF block count in a Morkov state
n_{sc}	Number of used subcarrier of OFDM
n_{sl}	The number of required states between two consecutive logs
$PL(dB)$	Path loss in dB scale
P_{ofdm}	Applied power in the OFDM block
P_{Tx}	Transmitter power
r	Remainder
T_s	Sampling interval
$x_{ch}(t)$	Channel block input signal

$x_{ofdm}(t)$	OFDM output signal
Δd	Distance difference
Δt_{log}	Time difference between two consecutive logs
θ	Phase
τ	Tap delay value
ϕ_l	Jakes' model random arrival times
ψ_l	Jakes' model random phases

LIST OF ACRONYMS/ABBREVIATIONS

1D	One Dimensional
2D	Two Dimensional
3D	Three Dimensional
acc	Accuracy
AoA	Angle of Arrival
API	Application Programming Interface
AWGN	Additive White Gaussian Noise
BPSK	Binary Phase Shift Keying
CCN	Convolutional Neural Network
COCO	Common Objects in Context
CP	Cyclic Prefix
dB	Decibel
DC	Direct Current
FFT	Fast Fourier Transform
FOV	Field-of-View
FPN	Feature Pyramid Network
GHz	Giga-hertz
GUI	Graphical User Interface
HD	High Definition
I	In-phase Component
IFFT	Inverse Fast Fourier Transform
IoU	Intersection over Union
IQ	In-phase and Quadrature Components
JSON	JavaScript Object Notation
KNN	K-Nearest Neighbors
mAP	Mean Average Precision
max	Maximum
MHz	Mega-hertz

MIMO	Multiple-Input and Multiple-Output
min	Minimum
OFDM	Orthogonal Frequency-Division Multiplexing
PAN	Path Aggregation Network
PL	Path Loss
Q	Quadrature Component
QPSK	Quadrature Phase Shift Keying
ReLu	Rectified Linear Unit
RF	Radio Frequency
RMSE	Root Mean Square Error
RNG	Random Number Generator
ROC	Receiver Operating Characteristic
SDR	Software Defined Radio
SiLu	Sigmoid Linear Units
SINR	Signal to Interference & Noise Ratio
SNR	Signal-to-Noise Ratio
std	Standard Deviation
STFT	Short-Time Fourier Transform
TDL	Tapped Delay Line
UAV	Unmanned Aerial Vehicle
USRP	Universal Software Radio Peripheral
XML	Extensible Markup Language
YOLO	You Only Look Once

1. INTRODUCTION

The drone industry is growing rapidly. According to Federal Aviation Administration statistics, as of 5 January 2024, 863,728 drones are registered in the US [1]. Also, in Turkey, registered drones became 68,426 in June 2023 [2]. In addition to using drones for good reasons, they can also be used for illegal uses like prohibited substance transportation, attacks for killing people, and more. Automated drone detection systems must be developed to protect certain areas from these types of illegal drone use.

Drones can be detected by different approaches. There are four main drone detection techniques in literature: vision, RF, acoustic, and radar-based solutions. Every method has advantages and disadvantages, as seen in Table 1.1. Building multi-model drone detection systems close to vulnerabilities of different techniques.

Several studies have built their own drone detection hardware architecture in the literature. However, drones and the hardware to detect them are inaccessible to every researcher. These researchers generally use publicly available datasets. However, these datasets are created for specific tasks like classifying drones and birds in images or classifying three drone brands by using RF signals. Also, created labels could not be sufficient for some tasks. We could not find any combined image and RF dataset in the literature.

To solve the dataset problem for drone detection research, we propose a drone image and RF dataset generation solution. In our solution, Unreal Engine is used as a simulation environment, and the Airsim plugin is used for a drone simulator. We developed an Airsim GUI, which communicates with the Airsim plugin in the Unreal Engine to easily create datasets. From the Airsim GUI, drones, cameras, RF, recording, and autonomous drone path settings can be configured for dataset generation.

1.1. Problem Definition

The thesis focuses on the problems associated with creating a dataset specifically designed to test drone detection models. One of the primary issues in building a drone detection system is the significant budget required to purchase drones and other necessary equipment. Also, configuring the system and making tests takes a lot of time. Furthermore, collecting synchronous data and manually labeling them is a burden for researchers, which can further increase the cost and duration of the project. A ready-to-use simulation system is needed to generate image and RF datasets for drone detection models.

1.2. Key Contributions

The thesis contributions can be summarized as:

(i) A digital twin of a drone detection system (with multiple data modalities) is implemented to ease the process of testing the drone detection algorithms.
(ii) Airsim GUI is created to communicate with the simulation environment, configure some settings, and generate a dataset.
(iii) USRP feature is contributed into the Airsim GUI to generate collected IQ signal.
(iv) By using the proposed image and RF dataset generation technique, three different drone detection models are proposed.

1.3. Thesis Outline

First, in Chapter 2, we review the literature about related topics. This chapter summarizes the related drone detection datasets and methods in the literature.

Chapter 3 presents overall simulation and dataset generation architecture. It includes an overview of all the technologies used, their configuration, and the Airsim GUI step-by-step dataset generation workflow.

Chapter 4 introduces all RF-related implementations. We also summarize the used channel model as a block diagram and explain every block.

Chapter 5 includes use cases for the Airsim GUI generated dataset. We provide a model for drone detection by using an image dataset and an RF dataset. Moreover, another model is provided for drone distance detection.

Finally, Chapter 6 summarizes the key contribution and provides a discussion on the possible future directions.

Table 1.1. Advantages and disadvantages drone detection methods.

Method	**Advantages**	**Disadvantages**
Camera	Cost-effective, provide visual evidence, etc.	Different scale and shape of drones, similarities with birds, different weather and lighting conditions, requires line-of-sight, etc.
RF	Resistance to different weather and lighting conditions, long range detection, no need for line-of-sight, etc.	Hard to differentiate from other RF resources, can not detect completely autonomous drones, etc.
Acoustic	Easy to deploy, cost-effective, no need for line-of-sight, etc.	A lot of background noise, short range, etc.
Radar	Long range, resistance to autonomous drones, localization, etc.	Hard to detect small drones, high cost, etc.

2. RELATED WORKS

2.1. Drone Detection Datasets

Some types of equipment are required for creating drone detection datasets. These pieces of equipment vary based on the created drone surveillance system, and drones want to be detected. Cameras, USRPs, microphones, and drones are mainly used for drone detection research. Because not everyone has enough budget, these pieces of equipment can not be reachable for everyone. However, there are several public datasets to use for free. These datasets are generally created for a single method, like drone detection with image or RF. Because our work covers only drone detection with camera and RF, we summarize the literature for these types of datasets.

2.1.1. Image Datasets

There are different workshops for drone detection with images. One of the most popular workshops is the Anti-UAV Workshop & Challenge. They create a dataset for challenges, and they are publicly published. This challenge happened three times. The first one happened in CVPR as the 1st International Workshop on Anti-UAV Challenge [3], and 160 video sequences, both RGB and infrared, were published. The 2nd anti-UAV Workshop & Challenge [4] happened in ICCV 2021, and they extended their dataset with 280 HQ videos. These videos are full HD thermal infrared video sequences and contain different-scale multiple drones. And the last challenge [5] happened in CVPR 2023. Again, the dataset was extended with more challenging thermal infrared video sequences.

Another popular challenge for drone detection is the Drone vs. Bird Detection Challenge. This challenge started in the first edition of the International Workshop on Small-Drone Surveillance, Detection, and Counteraction Techniques (WOSDETC) [6] at AVSS 2017 and happened five times. After it this challenge happen in 2019 at AVSS

[7], in 2020 [8], in 2021 at AVSS [9] and in 2022 at ICIAP [10]. The published datasets in these challenges are mainly for creating autonomous drone detection systems to differentiate birds and drones in different scales.

Another publicly accessible dataset published in [11]. Their main task is reconstructing the 3D flight trajectories by using multiple cameras. This supplementary dataset comprises five distinct sub-datasets, each featuring variations in the number of cameras, locations, and synchronization configurations.

2.1.2. RF Datasets

There are several RF datasets for drone detection in the literature. Authors of [12] shared one of the most popular RF datasets in the literature. It contains time series RF data recorded using universal software radio peripheral (USRP) in 2.4 GHz. Collected RF data contains Parrot Bebop, Parrot AR, and DJI Phantom 3 drones and only background RF activity. They also categorize the drone data into the drone modes as on, hovering, flying, and video recording. The dataset serves 2 classes for drone detection, 3 classes for drone type classification, and 10 classes for drone mode classification.

Authors of [13] and [14] shared datasets that contain raw IQ data. Authors of [13] shared a publicly available dataset DroneDetect V2. They created a dataset with drone signals with other signal interference in the same frequency band like Bluetooth. A BladeRF software-defined radio (SDR) and three different drone models are used. Created class types are similar to DroneRF dataset [12], and it has 2 classes for drone detection, 4 classes for drone type classification, and 10 classes for drone mode classification. The authors of [14] published a dataset containing remote controller signals and drones. After signal recording, they applied Labnoise and Gaussian noise to the drone signals. Also, they have a noise class that was generated using Labnoise and Gaussian noise.

The authors of [15] shared publicly available dataset called the Cardinal RF (CardRF) dataset. The main contribution of this dataset is that it contains line-of-sight and non-line-of-sight labeled drone RF data. Also, it contains different types of drones and a 2.4 GHz transmitters.

2.2. Drone Detection by Using RF Data

The DroneRF dataset [12] was used in different researches [16–21]. Authors of [16] published the paper for the generated dataset, and they used a deep neural network. The average accuracy is reported to be 99.7% for the 2-class problem, 84.5% for the 4-class problem, and 46.8% for the 10-class problem, respectively. Authors of [17] and [18] proposed 1D convolutional neural network and they achieve 59.2% and 87.4% accuracy in 10-class problem. Authors of [19] achieve 70.1% accuracy for the 10-class classification by using XGBoost, which is a more resource-friendly machine learning system than a deep neural network. Authors of [20] proposed a hierarchical model to improve the 10-class problem by using concatenated classifiers. First, these classifiers address a 2-class drone detection problem. Following that, if a drone is detected, they proceed to solve a 3-class drone type detection problem. Finally, the last set of classifiers is employed to tackle the task of identifying drone modes. Also, they used two different classification methods in their classifiers: XGBoost and KNN. They decided the classifier output by majority voting. And they achieve 99.2% accuracy for the 10-class problem.

In addition to sharing the dataset, authors of [14] also compares the performance metrics of drone detection by using IQ data and 2D spectrogram. They used the VGG network and demonstrated that a 2D spectrogram for drone detection is more robust when there are lower SNRs.

Authors of [22] proposed a drone detection system with a custom setup. They use two different SDRs and two different drones. They aim to localize the drone and its controller in the 3D domain by combining angle of arrivals (AoA) and distances. Their

proposed system obtains 12.2 degrees of error in identifying the drones' directions, 12.71 meters of error in localizing them, and the system obtains 9.9 degrees of error in identifying the controllers' directions, and 11.36 meters of accuracy in locating the controllers.

2.3. Drone Detection by Using a Camera

Like all other object detection tasks, the YOLO series is commonly used in drone detection. Authors of [23] used two different cameras for drone detection. One of them has a dynamic platform and lens for a low-angle view, and the other one has a static platform and lens for a wide-angle view. Their main approach is detecting objects in wide-angle view cameras and classifying them with low-angle cameras with 35X zoom. Thanks to this camera architecture, they can capture 6x6 pixels drones with YOLOv3. As a result, their false alarm rate is 0%, and the true positive rate is 91%. Authors of [24] used YOLOv4 for drone detection. They achieve a 74.36% mAP50 score using a dataset that contains a combination of bird and drone images. Authors of [25] used the latest version of YOLO, which is YOLOv8. Also, they modified the YOLOv8 and increased the performance metrics. They achieved an 85.2% mAP score by using the original version of YOLOv8s (YOLOv8 small model), and they improved it to 95.3% with the optimized version of YOLOv8.

Authors of [26] used the drone-vs-bird dataset, and they proposed a solution for static cameras. They split the drone detection problem into two phases: moving object detection and classification. They used the background subtraction method to detect moving objects and MobileNetV2 for classifying drones, birds, and backgrounds. For evaluation, they shared the F1-score as 0.742 for IoU is 0.3.

2.4. Drone Detection in Simulation Environments

Because of the reality of game engines, they are used for drone detection tasks. Drone detection via another drone camera is defined as a problem by authors of [27],

and they propose a solution using a depth map to solve this problem. They generate synthetic data in Unreal Engine and test the approach in real scenarios. The same team continues their study in [28]. Again, by using the same synthetic data generation approach, they improved their approach.

In the Drone-vs-Bird challenge in 2021 at AVSS [9], the winning team used a simulation environment to improve their score [29]. They implemented a 3D drone model and different 2D backgrounds imaged into the Unity game engine. Then, they recorded images from the game engine and merged them with the competition dataset. They achieved better results for drone detection with YOLOv5 by only adding synthetic data.

3. SIMULATION ENVIRONMENT SETUP FOR DRONE DETECTION

3.1. Simulation Architecture

We created a digital twin (simulator) of a drone detection system that uses image, RF, and other modalities. As the output, our digital twin produces output files for each modality and the ground truth labels by simulating the environment and entities. We especially focused on image and RF data generation systems in this thesis. However, we created our system to be as modular as possible to add different types of data generation processes like sound datasets for the future.

We can split our simulation architecture into three main parts: Unreal Engine as a game engine, GUI module, and the file system. We mainly developed the GUI part, which we called Airsim GUI, and we implemented small modules and configured other parts for our simulation system. These three parts interact with each other to generate desired measurement datasets.

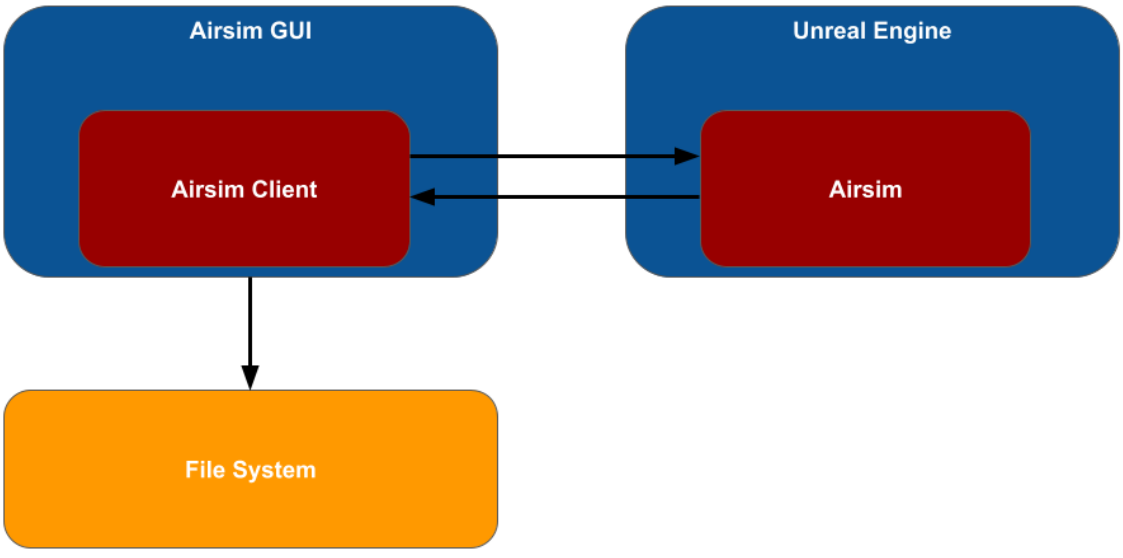


Figure 3.1. The simulation component architecture.

The simulation architecture is shown in the Figure 3.1. The Airsim component is an open-source simulator plug-in developed by Microsoft for drones in Unreal Engine that eases the implementation of drone-related operations. Airsim has an Airsim client library, and we used it in the Airsim GUI to communicate with the Airsim in the Unreal Engine. As a result of communication, we can manage and interact with the simulation environment, take actions, and collect required data in the Airsim GUI. Based on the simulation configuration, the generated dataset files are written into the file system during and after the simulation.

3.2. Used Technologies

Here in this Section, we will elaborate on the technologies that are used in the drone detection system simulator.

3.2.1. Unreal Engine

To create the environment and generate datasets in the digital twin, Unreal Engine 4 is chosen as a simulation platform. The Unreal Engine is a widely used game engine in the gaming community, and its main competitor is Unity. Also, there are different game engines like Godot and CryEngine. We used Unreal Engine rather than Unity because of the more realistic vision and built-in Airsim compatibility. The description of Airsim will be mentioned in the following subsection, but basically, it provides drone functionalities in the Unreal Engine.

We need to understand Unreal Engine's terminology to understand the implemented parts.

- Project: It contains all of the files and folders in the created game.
- Blueprint: Unreal Engine supports a no-code development environment in addition to C++ support. The Blueprint Visual Scripting system contains nodes, events, functions, and variables like a box. These boxes are connected to create

flow. These boxes can refer to a reference of an object, an action in the keyboard, or different functionalities. A blueprint example from our simulation environment is shown in Figure 3.2.

- Object: It is a fundamental unit in Unreal Engine. In Unreal Engine, almost everything inherits from an object.
- Level: Created sections of a game are called levels in Unreal Engine. This level can be called a simulation environment in our case. Different types of objects and functionalities can be implemented at different levels.
- Actor: It is the component that can be placed into a level. Lights, cameras, and static meshes are some of the actors.
- Pawn: It is a subclass of actor. Pawns are characters that are controlled by users or artificial intelligence on the level. Drones are an example of pawns in our project.

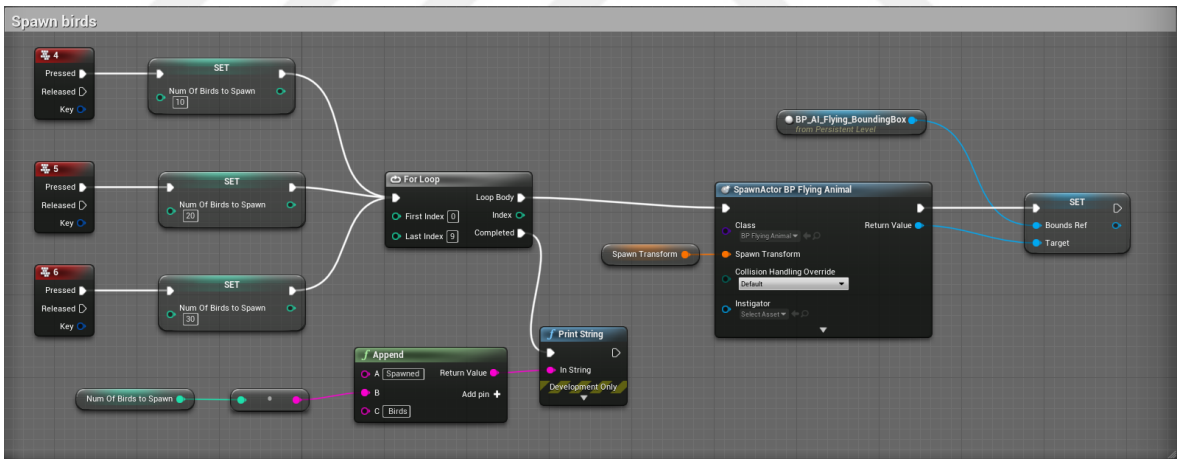


Figure 3.2. Blueprint example: adding actions to keyboard commands for adding birds to the simulation environment.

3.2.2. Airsim

Airsim is an open-source cross-platform simulator for cars and drones. It was developed by Microsoft. It is mainly used for autonomous technologies. Airsim can be added as a plugin into an Unreal Engine project. Also, there is an Airsim client library

for Python to send requests to Airsim from outside of the Unreal Engine project. In Unreal Engine, Airsim works as a server, and clients use the Airsim library.

We mainly used the drone and camera functionalities of Airsim and Airsim client to communicate with Airsim. We send drone actions and collect data by using the Airsim client library.

3.2.3. Pyphysim

Pyphysim is a Python library that provides simulation for digital communication physical layer. It has different classes offering digital modulation techniques like BPSK and QPSK, different types of channel models, and MIMO. This library is created with a modular approach therefore, the communication process can be built by concatenating the functions, and new functionalities can be added to this library. We used pyphysim in the RF data generation steps.

3.2.4. PyQt

To make data generation steps easier for researchers we designed a GUI for them. We chose PyQt for creating GUI. PyQt is a cross-platform GUI designer tool for Python. It is binding for Qt which is a C++ framework. Because we used Python in almost every part of the project, we chose a GUI solution in Python.

PyQt has a designer tool to make designing GUI easier. The view of the PyQt designer with our project is shown in the Figure 3.3. PyQt has a wide range of widgets, and they provide different functionalities. For example “Line Edit” is used to take input from the user and “Push Button” is used for emitting the click button. Widgets are located in the GUI with drag and drop by using the designer tool. A unique object name must be defined for every widget. After finishing designing the GUI in the designer tool “.ui” formatted file is generated automatically. This “.ui” formatted file contains all information about GUI and widget attributes in XML markup language.

Then this file is transformed into a Python file by using PyQt script. These created widgets can be accessed from different Python scripts by using unique object names.

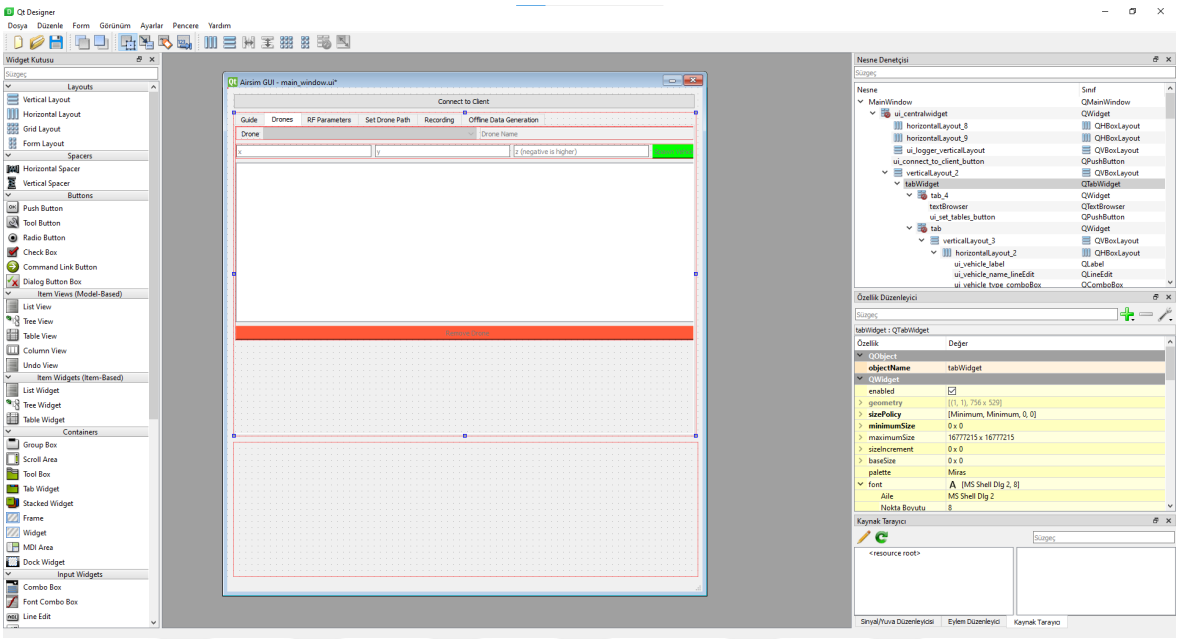


Figure 3.3. PyQt designer.

3.3. Unreal Engine Level Design

We chose some areas of the Boğaziçi University north campus as a simulation environment for our test scenarios. Our environment covers the library, computer engineering building, basketball court, portal statue, and the pyramid. This area covers approximately $10\,300\,\mathrm{m}^2$. The selected area's map image is in Figure 3.4. The modeled of Aptullah Kuran library which is shown in Figure 3.5 will be demolished and rebuilt because of the building risk report [30].

We can split the level design part into two phases: creating materials and combining these materials. In creating the materials part, we created chosen buildings for the simulation environment. In the combining part, we made the selected area in Unreal Engine by using the materials created in the first phase.

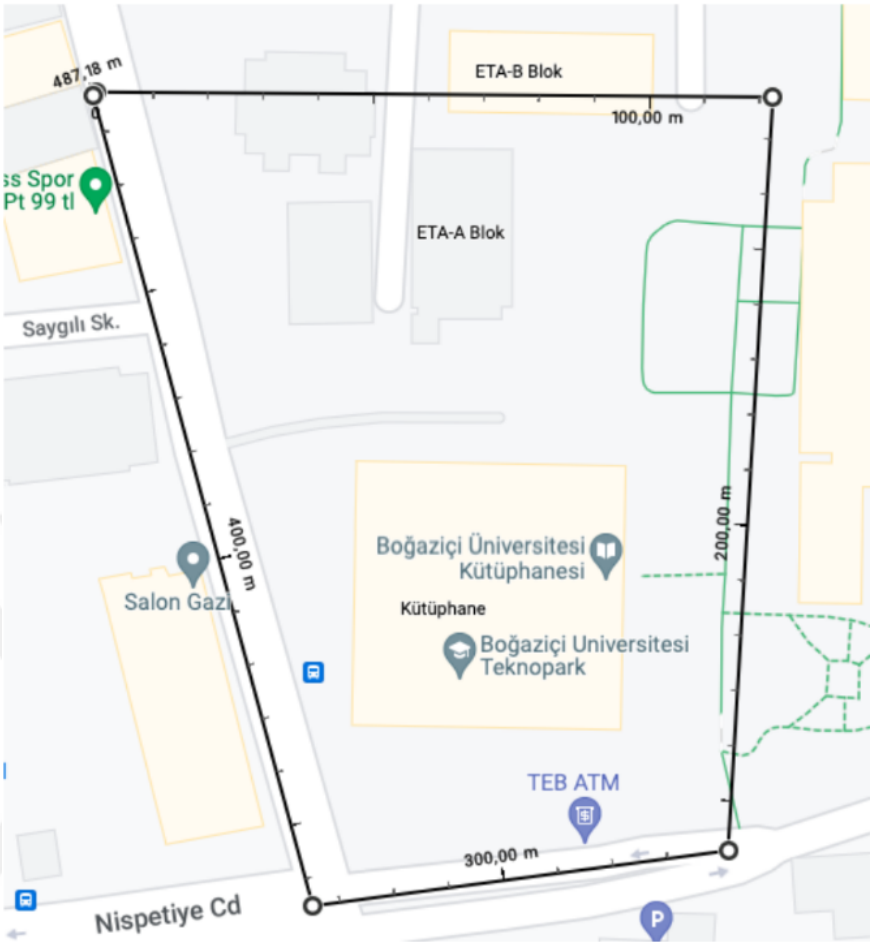


Figure 3.4. Map image of the selected simulation environment.



Figure 3.5. Aptullah Kuran Library.

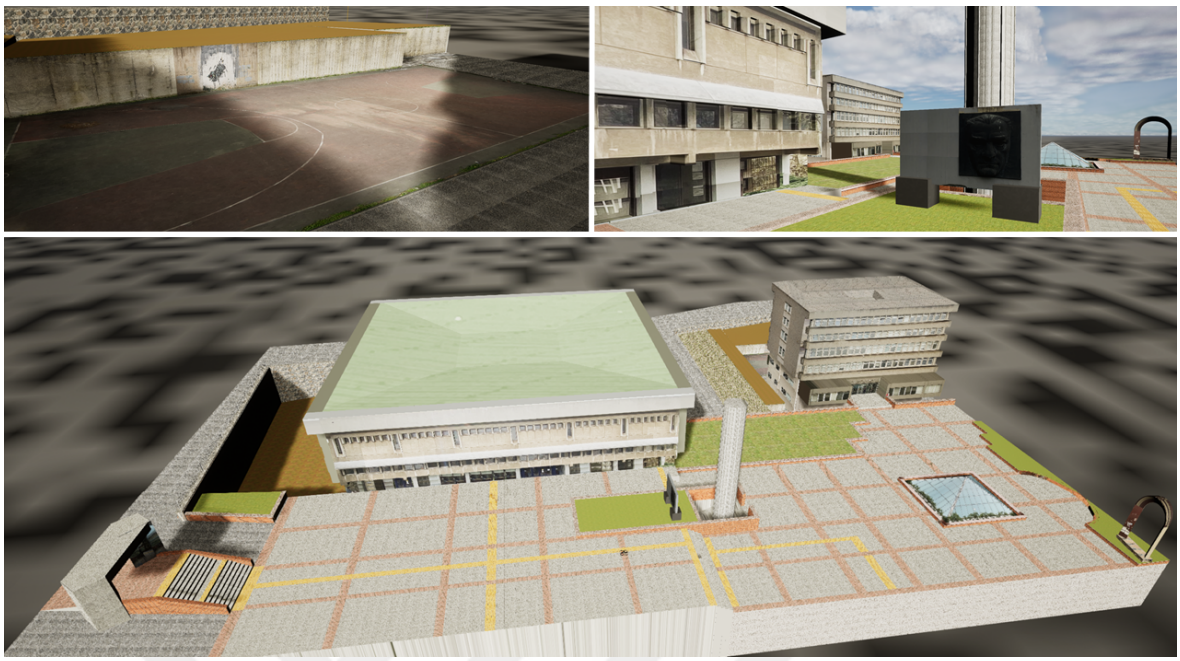


Figure 3.6. Screenshot images from the simulation environment. Top left image: the basketball court. Top right image: the Atatürk bust. Bottom image: the campus view from East.

In the first phase, we used DJI Mavic Air 2 as a drone to take pictures of the buildings we selected. We scanned all areas of the buildings with the drone. We combined the inspected areas using Blender. Then, we created the buildings in one piece with a 1:1 scale for the simulation environment.

In the second phase, we exported the created materials in the Blender and moved to the Unreal Engine project. Then, we combined these materials in the Unreal Engine level. As a result, we built the chosen area in the Unreal Engine with a 1:1 scale. Images from the simulation environment are shown in Figure 3.6.

We also added birds to the simulation environment. A free bird blueprint is used [31]. There is a bird and a flying bounding box object. The flying bounding box object limits the bird's flying area. Multiple birds can be added to this box. In our simulation environment, we defined actions for the "4", "5" and "6" buttons to add more birds with a level blueprint as seen in Figure 3.2.

Also, we define actions to change simulation hours. The “7” button makes the environment daytime and the “8” button makes it nighttime. This functionality is also implemented by using a blueprint.

3.4. Airsim GUI Development

3.4.1. Airsim GUI

For data generation in Unreal Engine, we created a GUI that communicates with the drone detection system in the simulation environment. We named this GUI as the Airsim GUI. Communication between Airsim GUI and Airsim is performed by using the Airsim client library for Python. The client library has several APIs for several tasks. These APIs use an inter-process messaging library in Python. The name of the messaging library is msgpack-rpc-python.

For designing the GUI, we used PyQt and its designer tool. As described in Subsection 3.2.4 “.ui” formatted file is created for GUI design. We have a single “.ui” file for the project. We gather all functionalities in this “.ui” file. The main screen has three different sections. The first section covers only the “Connect to Server” button. This button provides a connection to the running Unreal Engine project that has the Airsim plugin. The second section covers the tab widget. This tab widget has different types of tabs categorized based on different functionalities. For example, functionalities about defining drone paths for autonomous driving are located in the “Set Drone Path” tab, and configurations of RF data generation are located in “RF Parameters”. Finally, the last section in the GUI is the logging part. Digital twin logs of the drone detection system are displayed in this part.

3.4.1.1. Connect to Server. Airsim GUI can be used for real-time and offline data generation. For real-time data generation, Airsim GUI needs to communicate with Airsim in the Unreal Engine. Before establishing the connection between Airsim GUI and Unreal Engine, some of the configurations need to be completed.

```
"Tables": {
    "Table1": {
        "X": -50,
        "Y": -40,
        "Z": -0.02,
        "Rotation": 0
    }
}
```

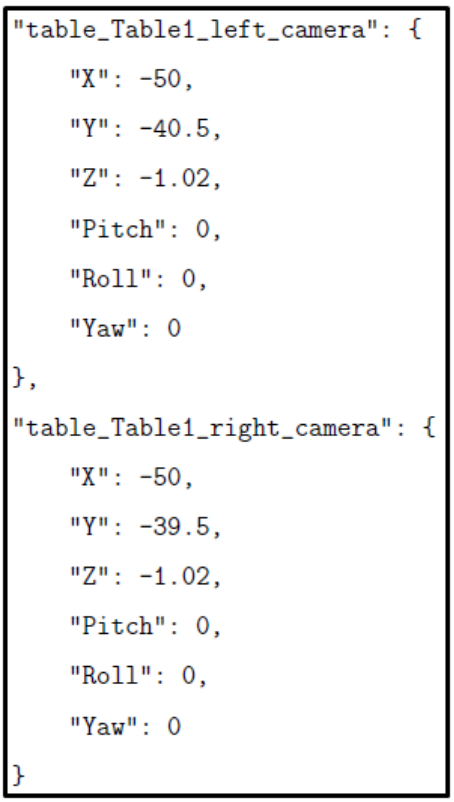
Figure 3.7. Table settings.

Airsim reads configuration from the "settings.json" file from the file system before running the Unreal Engine level. The location of the dataset, spawned drones, and cameras are configured from this file. We also added a table object in this configuration file. We created the table object, because we wanted to use two cameras with a fixed distance between them. Changing two different cameras' locations and angles separately based on this camera configuration needs mathematical calculations. However, by using table objects after setting only one coordinate and rotation, two different cameras can be located at a fixed distance, rotated to the same angle, and centered on the entered coordinates. For example in the JSON shown in Figure 3.7, a single table is located in the (-50, -40, -0.02) coordinate in the Unreal Engine world. And its rotation is 0. There are four different rotations, and every rotation refers to different angles.

There is a button to prepare the "settings.json" file on the first screen of GUI. After clicking this button, it creates two cameras in the settings file as shown in Figure 3.8. These cameras are called external cameras and they are not connected to any of the drones.

After clicking the "Connect to Server" button, the GUI connects to Airsim in the Unreal Engine instance and then spawns the table in the given location. After establishing the connection, we retrieve some information about the simulation environment

and the configuration, like camera locations, drone names, and the dataset creation directory. These pieces of information are used during the recording.



```
"table_Table1_left_camera": {
    "X": -50,
    "Y": -40.5,
    "Z": -1.02,
    "Pitch": 0,
    "Roll": 0,
    "Yaw": 0
},
"table_Table1_right_camera": {
    "X": -50,
    "Y": -39.5,
    "Z": -1.02,
    "Pitch": 0,
    "Roll": 0,
    "Yaw": 0
}
```

Figure 3.8. Located cameras based on configured table settings.

3.4.1.2. Tab Widget. In the tab widget, there are different tabs based on their functionalities:

- Guide: It includes how to use GUI. It also has "Prepare the settings.json File" described in the Section 3.5 in detail.
- Drones: This part has spawning and deleting drone functionalities in running Unreal Engine instance. When spawning a drone, the drone name and XYZ coordinates are taken as input. For deleting a drone, the drone name is selected from the list.
- RF Parameters: In this part RF specific parameters are managed. These parameters are described in Chapter 4.

- Set Drone Path: In this tab, different paths can be planned for every drone. After defining paths, drones can follow the paths autonomously. The given paths can be saved and loaded.
- Recording: In the recording tab, mainly recording functionalities are managed. Initially, a user selects which cameras will be recorded in which format. Then, fill or leave the checkboxes empty for recording functionality. These checkboxes are "Save segmented images", "Follow path while recording", and "Generate RF after recording". "Save segmented images" allows for recording drone segmentation while collecting camera images. "Follow path while recording" ensures that after clicking the "Start Recording" button, drones will follow the given paths in the "Set Drone Path" tab. If the "Generate RF after recording" option is selected, RF data generation (with the configured RF parameters) starts after the simulation scenario finishes in the Unreal Engine.
- Offline Data Generation: Because the RF generation step takes a long time, for the same scenario, RF generation can be done with different parameters. We created the "Offline Data Generation" tab to execute this process by considering the simulation logs. In this tab, RF data can be generated by providing a specific recording folder without connecting and doing the flight simulation again.

3.4.1.3. Logging. In the logging section, we display GUI-specific logs. We are capturing the Python-specific logs and we are creating logs for our processes. This logging part provides debugging logs and following triggered actions.

3.4.2. Airsim GUI Recording Structure

For recording the simulation output, we have four different functions that run in different Python worker threads. These functions are categorized as requesting and recording. Requesting functions are responsible for collecting data from Unreal Engine, and recording functions are responsible for writing collected data into the file system. Between requesting and recording functions, there is a queue data structure for communication. Every requesting function matches with only one recording func-

tion. Splitting requesting and recording functionalities prevents the slower requesting because of slow recording performance.

The first requesting-recording function pair is created for recording drone-specific logs. When recording is started, the "vehicle_logs" folder is created in the parent recording folder, and for every drone a new file is created. We are logging timestamp, Cartesian coordinate position, and Cartesian coordinate velocity.

The second requesting-recording function pair is responsible for recording image data. Only the views of the selected cameras in the "Recording" tab are recorded for the dataset. During the recording with a single request, all selected cameras' images are collected in a single timestamp. These images are sent to the recording function by using the communication queue between the requesting and recording functions. Finally, the recording function writes images and their information to the file system. A new folder is created to store information about the timestamp and image types of the recorded images. The created directory name is "camera_logs". For every camera, a single file is created with the camera name to store the image timestamps, image types, and image paths. In addition to the "camera_logs" folder, new folders are created with the camera names to save images into these folders.

After a recording finish, the created folder hierarchy for dataset is shown in Figure 3.9.

3.4.3. Airsim GUI RF Data Generation

We mainly used pyphysim library in Python to generate the RF data. The created RF dataset refers to collected IQ signals by USRPs in the simulation environment. The RF data generation system supports multiple drones and USRPs. For this version of our data generation system, drone controllers are excluded since they are assumed to be out of coverage, as seen in Figure 3.10. Because of that, only drone-transmitted data is generated. USRP, drone transmitters, and other RF simulation configurations

are set in the “RF Parameters” tab of Airsim GUI. Then, the RF dataset can be created directly after recording or using previously created datasets in the “Offline Data Generation” tab. For offline data generation, the created “vehicle_logs” folder must be given manually.

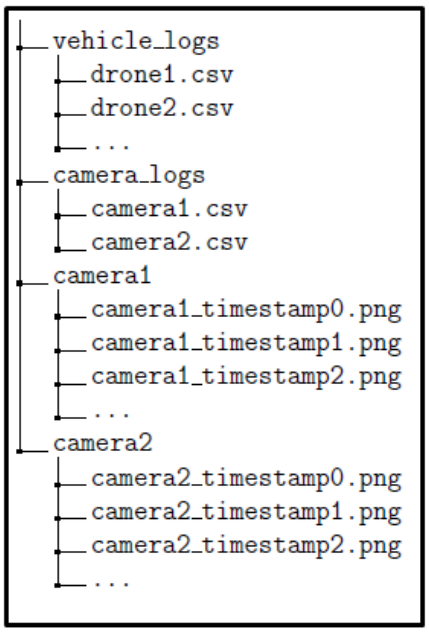


Figure 3.9. Example of generated vehicle and image dataset folder hierarchy.

High-level RF dataset generation steps are shown in Figure 3.11. Initially, the generated drone logs are read under the “vehicle_logs” folder. For every drone, data transmission or not transmission state is chosen based on the Markov chain. If the drone generates RF signals in a state duration, the drone transmitter generates random symbols and modulates them with BPSK or QPSK over OFDM symbols. These modulated signals pass through the channel. We implemented Jakes’ and tapped delay line channel models. Then distance related effects like path loss and phase shift for USRPs are added. After these effects are added, each drone signal for a specific USRP is summed together. At the end the AWGN channel is added to the signal and USRPs receive these signals. These received signals are written to a file system as an IQ signal. Also, the used RF parameters are written into a file system. The generated files are written into the “rf_logs” folder as in Figure 3.12.

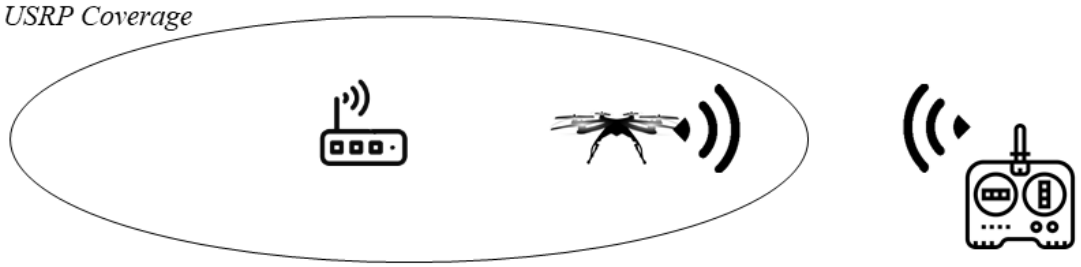


Figure 3.10. The USRP coverage in our simulation system shows that drone controllers are always out of coverage.

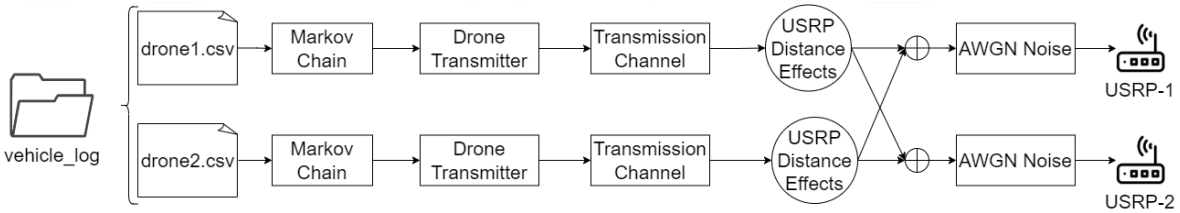


Figure 3.11. High-level RF data generation steps.

3.5. Airsim GUI Workflow

In this section, we will be clarifying the steps to execute a drone detection simulation in the digital twin of the drone detection system. The steps that we require are as follows: Unreal Engine environment setup, Airsim configuration, preparing the table of the detection system, running the Unreal Engine level, connecting to the client, adding additional drones, setting the RF parameters, setting the drone path, recording the experiment, and offline data generation.

```
└─rf_logs
  ├─rf_parameters.pickle
  ├─rf_iq_data_usrp1.pickle
  ├─rf_iq_data_usrp2.pickle
  └─...
```

Figure 3.12. Example of generated RF dataset folder hierarchy.

3.5.1. Unreal Engine Environment Setup

We created a 3D model of the Boğaziçi University north campus as the simulation environment in Unreal Engine as described in Section 3.3. Users can change this environment easily, a new Unreal Engine project can be created and Airsim needs to be added to the project to change the 3D world of the digital twin.

3.5.2. Airsim Configuration

Settings file is used for the configuration of Airsim in the Unreal Engine. Airsim requires this file in the "Documents\AirSim" folder with the "settings.json" name. Before running the Unreal Engine level, this file needs to be prepared. Airsim reads this file only once when the Unreal Engine level starts.

Airsim supports both cars and drones for simulation. Because of that, the drone simulation option needs to be selected as in Figure 3.13. Configuring "ViewMode" as "SpringArmChase" enables the chasing camera to view the drone perspective. However, it is not an important setting in our case because we are creating routes from GUI and drones move autonomously.

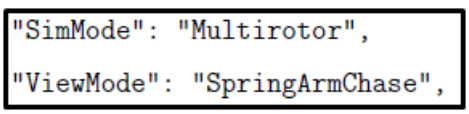


Figure 3.13. Airsim "SimMode" and "ViewMode" settings.

We are using the recording configuration of Airsim. Normally, there is only a camera logging feature in Airsim, but we do not use it directly. Therefore, configuring the recording folder is enough for our dataset generation system as in Figure 3.14.

```
"Recording": {
    "Folder": "D:/AirsimSavedData",
}
```

Figure 3.14. Airsim "Recording" setting.

"Subwindows" in the configuration provides displaying specified cameras in the simulation screen as in Figure 3.16. This feature is good for debugging the cameras; however, it decreases the logging frequency dramatically. Subwindows can be closed by pressing the "0" button. Subwindows can be configured as in Figure 3.15.

```
"SubWindows": [
    {
        "WindowID": 0,
        "ImageType": 0,
        "CameraName": "front_center",
        "Visible": true,
        "External": false,
        "VehicleName": "SimpleFlight"
    }
]
```

Figure 3.15. Airsim "SubWindows" settings.

Airsim provides different types of camera configuration in Unreal Engine. These camera types are chosen with an enumerator. The mapping of enumerators and camera types is shown in Figure 3.17(a).

By configuring "CameraDefaults", we can manage the default camera settings of different camera types. We can change the field-of-view (FOV), width, and height settings of cameras. An example of changing the default settings of "Scene" view is shown in Figure 3.17(b).

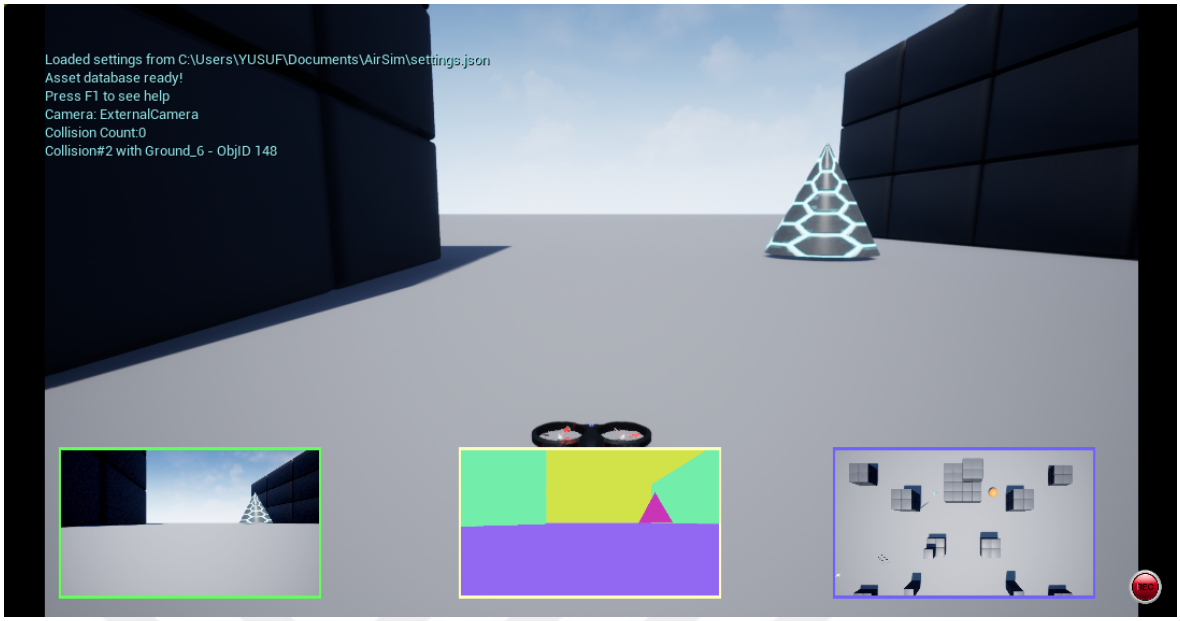


Figure 3.16. Unreal Engine with Airsim. Three small images at the bottom of the figure are subwindows in Airsim.

We can configure the deployed vehicles in the simulation environment. Also, we can deploy configured vehicle types after running the Unreal Engine from the Airsim GUI. We used the "SimpleFlight" type in Airsim because we do not need autonomous drone system hardware simulation. An example of our used vehicle configuration is shown in Figure 3.18(a). We need to set "AutoCreate" and "AllowAPIAlways" as true. After Airsim GUI connects Airsim, it sends API requests to this drone. The deployment coordinates and rotation can be configured by using "X", "Y", "Z", "Pitch", "Roll", and "Yaw" variables. Also, we can add cameras to the drones in this configuration.

Airsim cameras that are not connected to drones are called external cameras. We are using this type of camera for different purposes. The first purpose is to capture the simulation environment from a bird's eye view. Using this view, we can set drone routes in the following steps. The other purpose of using these camera types is to build drone detection systems for detection by using cameras. We mainly manage these cameras by using a table configuration that will be described in the next step. However, a user can add cameras manually. The "map-camera" configuration needs to be copied from Figure 3.18(b) directly. Because Airsim GUI directly connects "map-camera" for

route planning. This camera looks towards the ground in the middle of the simulation environment at an altitude of 150 meters. In addition to this camera, different cameras can be added from this setting.

```
Scene = 0,
DepthPlanar = 1,
DepthPerspective = 2,
DepthVis = 3,
DisparityNormalized = 4,
Segmentation = 5,
SurfaceNormals = 6,
Infrared = 7,
OpticalFlow = 8,
OpticalFlowVis = 9
```

(a)

```
"CameraDefaults": {
    "CaptureSettings": [
        {
            "ImageType": 0,
            "FOV_Degrees": 90,
            "Width": 1280,
            "Height": 720
        }
    ]
}
```

(b)

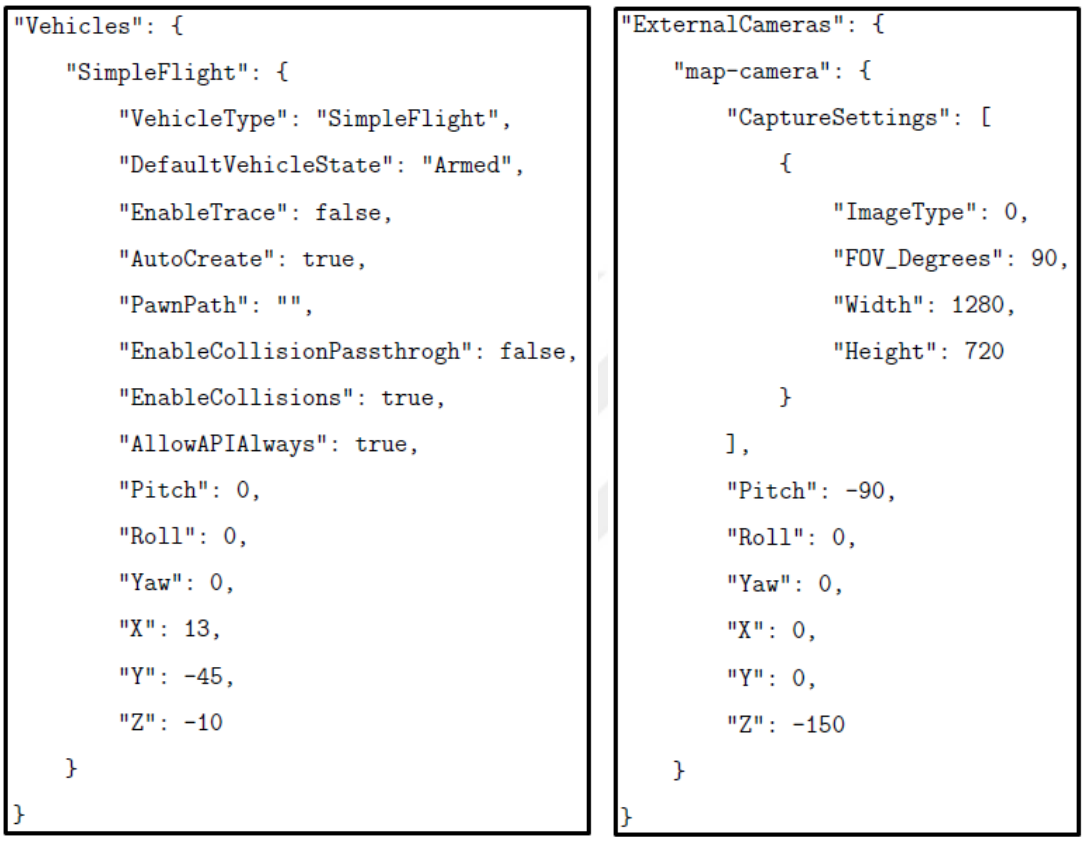
Figure 3.17. Camera types and default camera settings in Airsim. (a) Supported camera types with enumerators in Airsim, (b) an example of changing the default settings of “Scene” view.

As described in Section 3.3, we created another object type, and it is called table. We created it to manage multiple cameras with the same configuration for different scenarios. We created tables with two different cameras, and there is one meter between them. We can also manage the rotation of the table. There are four different rotation supports for tables, and every neighbor rotation has 90 degrees between them. An example of table configuration is shown in Figure 3.7.

3.5.3. Preparing the Table of the Detection System

After completing the Airsim setting file configuration, press the “Prepare the settings.json File” button in the “Guide” tab from Airsim GUI. The “Guide” tab is

shown in Figure 3.19. This button reads the table configurations, deletes old tables' external cameras, and then creates new tables' external cameras. The created cameras are shown in Figure 3.8.



```
"Vehicles": {
    "SimpleFlight": {
        "VehicleType": "SimpleFlight",
        "DefaultVehicleState": "Armed",
        "EnableTrace": false,
        "AutoCreate": true,
        "PawnPath": "",
        "EnableCollisionPassthrogh": false,
        "EnableCollisions": true,
        "AllowAPIAlways": true,
        "Pitch": 0,
        "Roll": 0,
        "Yaw": 0,
        "X": 13,
        "Y": -45,
        "Z": -10
    }
}
```

(a)

```
"ExternalCameras": {
    "map-camera": {
        "CaptureSettings": [
            {
                "ImageType": 0,
                "FOV_Degrees": 90,
                "Width": 1280,
                "Height": 720
            }
        ],
        "Pitch": -90,
        "Roll": 0,
        "Yaw": 0,
        "X": 0,
        "Y": 0,
        "Z": -150
    }
}
```

(b)

Figure 3.18. (a) Airsim "Vehicles" settings, (b) Airsim "ExternalCameras" settings.

3.5.4. Running the Unreal Engine Level

Until now, all required Airsim configurations have been set. When we start the prepared Unreal Engine Level, Airsim reads the configurations and prepares the environment. All cameras and drones are located based on the "settings.json" file.

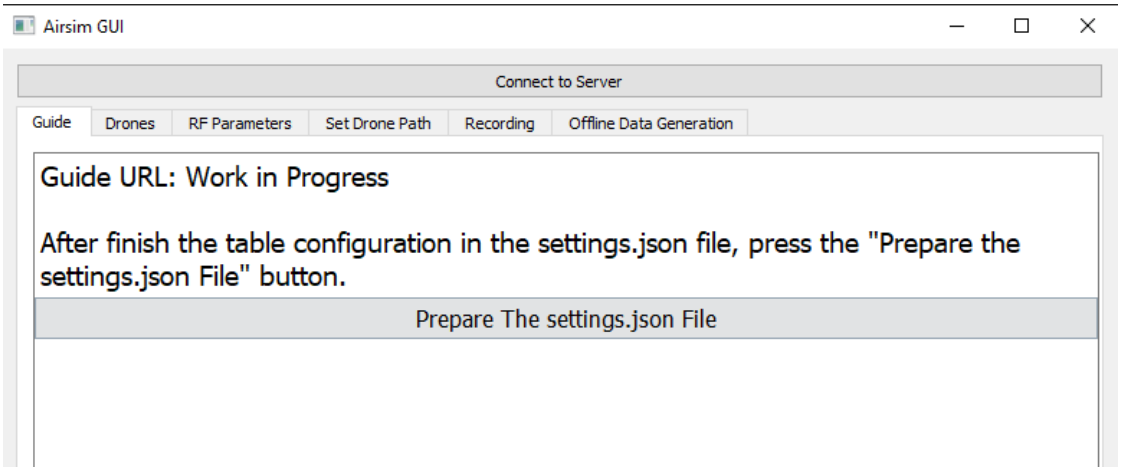


Figure 3.19. Airsim GUI "Guide" tab.

3.5.5. Connecting to the Server

When we press the "Connect to Server" button in the Airsim GUI, a connection is established between the Airsim GUI and the Airsim in the Unreal Engine. This main connection is persistent but for different functionalities non-persistent connections are established.

3.5.6. Adding Additional Drones

We spawned drones by using the "settings.json" file in the previous steps. However, we can spawn different drones from the Airsim GUI with prepared configurations in the "settings.json" file. The "Drones" tab needs to be opened as in Figure 3.20. A unique drone name and spawn location need to be provided. Then the drone can be spawned by pressing the "Spawn Vehicle" button. Also, spawned drones can be deleted by choosing them in the drone list in the Airsim GUI.

3.5.7. Setting the RF Parameters

If a user wants to generate RF data, all RF parameters need to be filled. These fields are shown in the Figure 3.21. The RF data generation is described in Chapter 4.

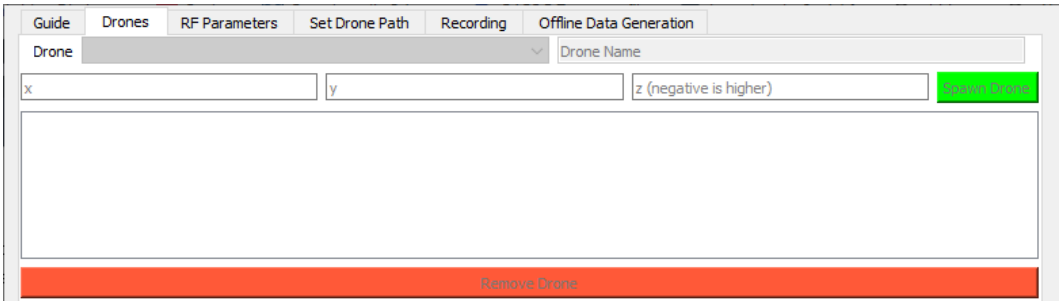


Figure 3.20. Airsim GUI "Drones" tab.

We need to fill OFDM parameters with integer values. They are standard OFDM parameters, and they are FFT size, number of used subcarriers, and cyclic prefix. For Jakes's channel model, we only have one parameter and it is the number of rays. Also, this value needs to be an integer. As a path loss model, we used the free space path loss model and this model accepts two different parameters from the user. They are the path loss coefficient and carrier frequency. Both of them need to be a float and the carrier frequency unit is MHz.

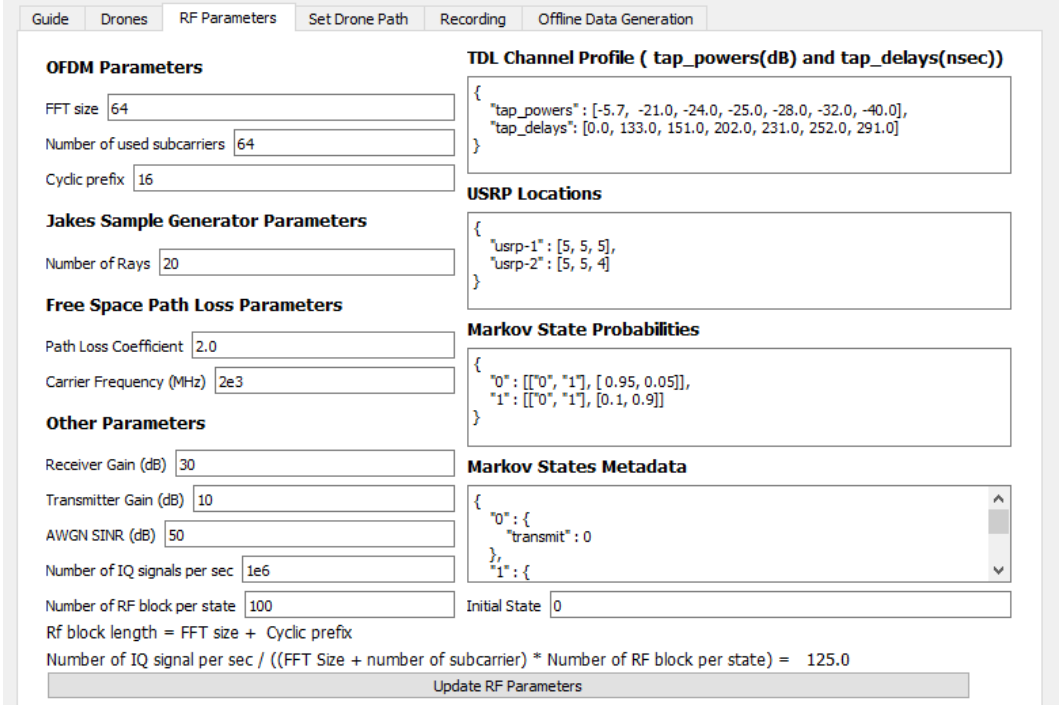


Figure 3.21. Airsim GUI "RF Parameters" tab.

```
{
    "tap_powers" : [-5.7,  -21.0, -24.0, -25.0, -28.0, -32.0],
    "tap_delays": [0.0, 133.0, 151.0, 202.0, 231.0, 252.0]
}
```

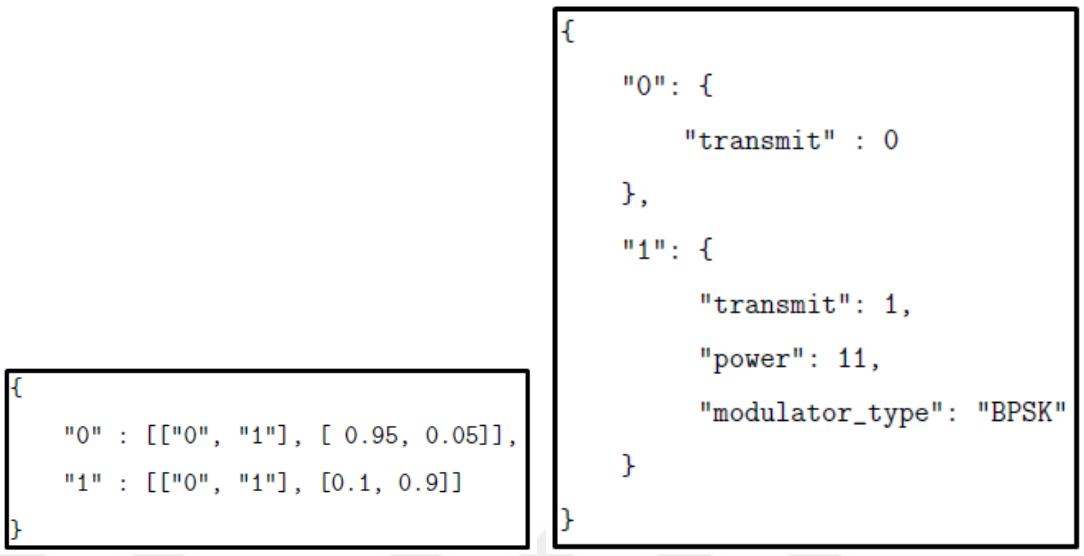
Figure 3.22. An example of TDL channel settings.

```
{
    "usrp-1" : [5, 5, 5],
    "usrp-2" : [5, 5, 4]
}
```

Figure 3.23. An example of USRP settings.

Under the other parameters, there are parameters used in different parts of RF data generation. Transmitter and receiver gains are used to increase the strength of the signal. Their units are dB. We are using the AWGN channel for noise, and its noise parameter is taken in the dB unit. The number of IQ signals per second defines the sampling rate of a received signal. If a user fills this value with 10^6 the sampling rate will be 1 nanosecond. We also used Markov model with two states for traffic generation. The number of RF block generations per state defines how many symbols will be generated during a single Markov state. The calculation of it is described in Chapter 4.

In the right panel of the RF parameters, mainly JSON formatted inputs are taken. First, the TDL channel profile is filled with tap powers and tap delays information of the channel. These two arrays must be filled with the same length. The tap powers unit is dB and the tap delays unit is a nanosecond, and their values are float. An example of the TDL channel which has six taps shown in Figure 3.22.



(a) (b)

Figure 3.24. An example of Markov chain probability and metadata settings. (a) Markov chain probabilities, (b) Markov chain metadata.

USRP locations are set by describing the unique name of the USRP and the values of its coordinates. There is no limitation for the number of USRP, but a higher number can cause memory and speed problems in the RF generation. An example of locating two different USRP is shown in Figure 3.23.

We used Markov chain for RF signal transmission or not transmission choice. Markov chains have states and these states have probabilities for the next state and metadata about RF signals. For state probability, a two-dimensional array needs to be created for every state. The first row of the arrays describes which states can be passed after the current state, and the second row shows the probabilities of passing these states. Markov states metadata mainly shows whether drones generate RF or not, and if data will be generated, the power and the modulation technique are written into metadata. For every state, probabilities need to be set and metadata needs to be filled. Finally, the state from which the simulation will start is given in the initial state input. An example of Markov chain probability and state metadata JSON files is given in Figure 3.24.

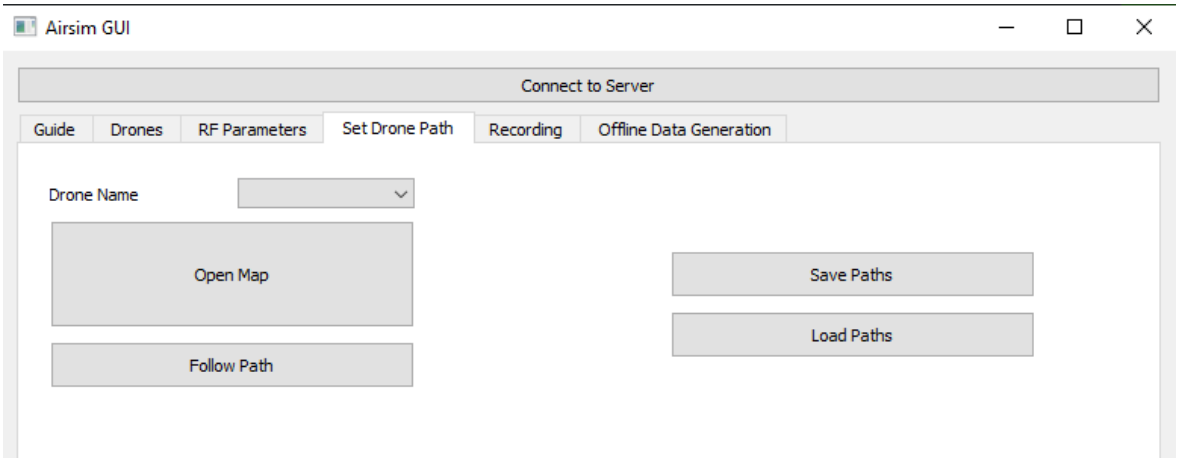


Figure 3.25. Airsim GUI "Set Drone Path" tab.

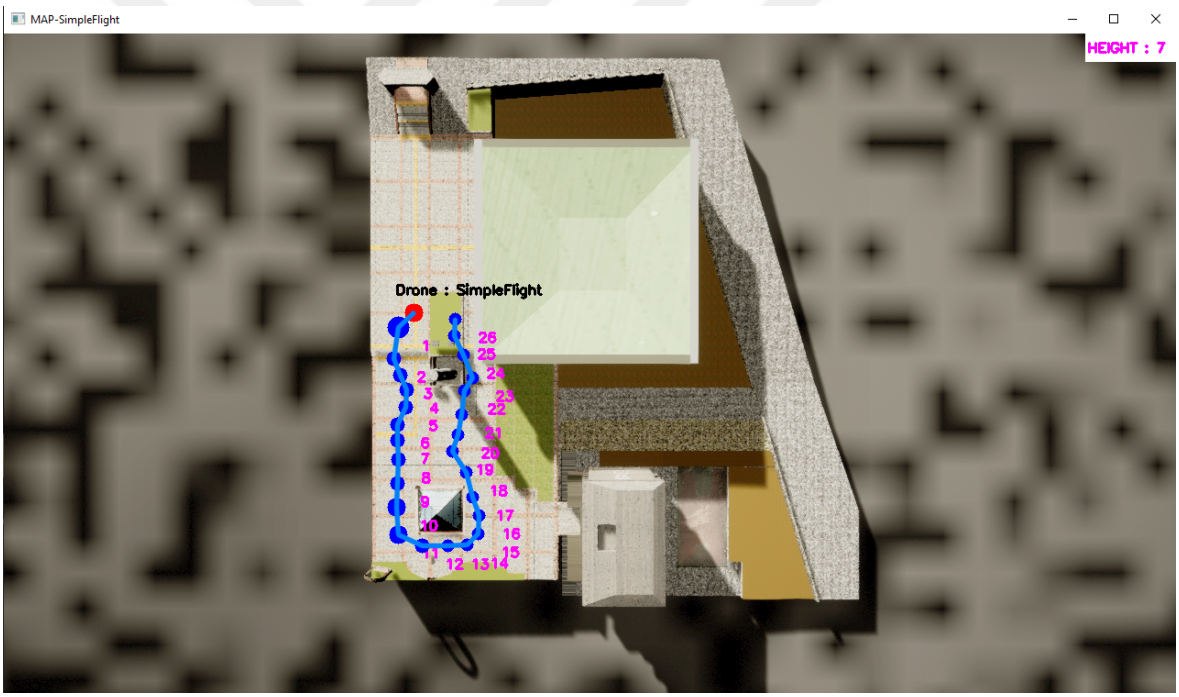


Figure 3.26. Route planning example in the Airsim GUI.

3.5.8. Setting the Drone Path

The drones' routes can be set in the "Set Drone Path" tab. The "Set Drone Path" tab is shown in Figure 3.25. First, the drone is selected from the drone names combo box. Every spawned drone is shown in it. After choosing the drone, click the "Open Map" button. In the open map, the checkpoints can be placed with the altitude.

Altitudes are adjusted by scrolling down and up. The distance between checkpoints needs to be close as in Figure 3.26. The altitude information is located at the top right. After setting the path for a drone, press the "Q" button to exit the route planning window. After finishing the route planning, paths can be saved. The paths are saved in the "Documents/Airsim" folder as "saved_paths.json". When pressing the "Load Paths", Airsim GUI searches this file in the same path. After setting the drone path is finished, drones can start following the route after clicking the "Follow Path" button. Also, it can be configured as drones follow the path when recording starts.

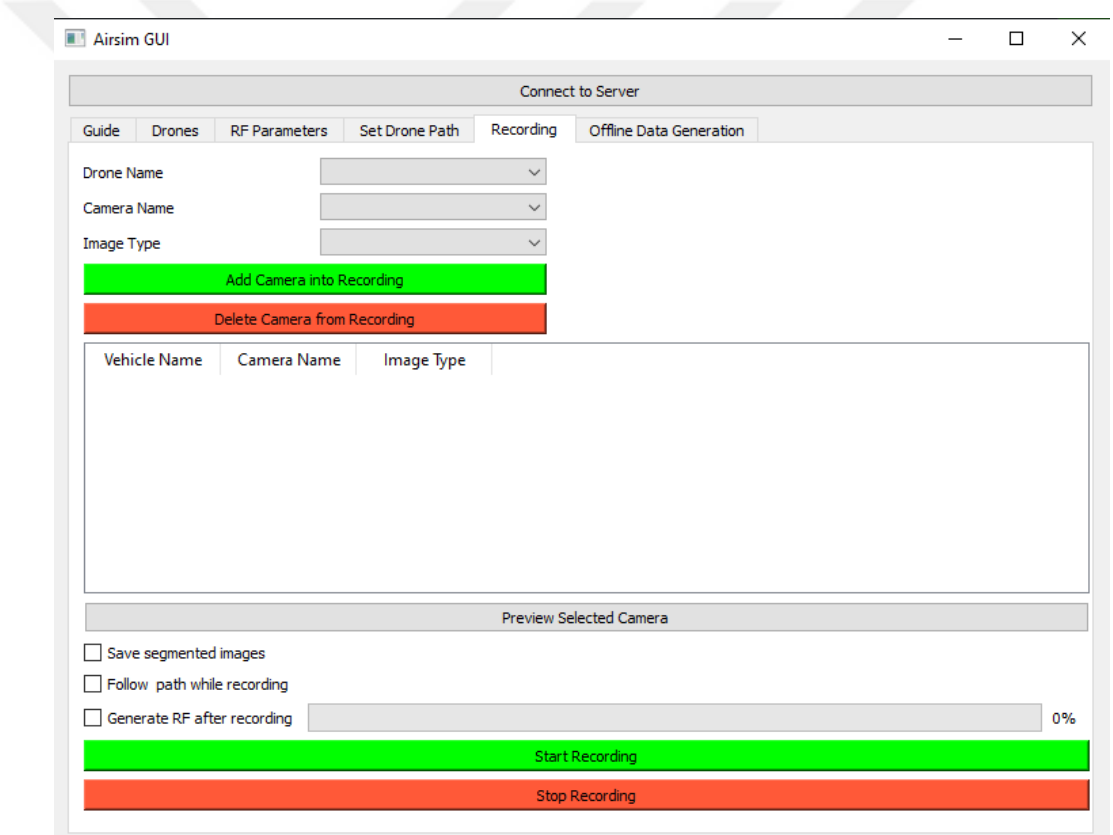


Figure 3.27. Airsim GUI "Recording" tab.

3.5.9. Recording the Experiment

The "Recording" tab is shown in Figure 3.27. In the "Recording" tab, which camera will record with which mode can be selected and removed from the list. To choose external cameras for recording, first, a drone needs to be selected. The se-

lected external camera is not related to the chosen drone. After selecting the recording cameras, checkboxes need to be filled or not filled.

"Saved segmented images" provide create and save segmentation of drones for selected cameras. An example of a segmented image is shown in Figure 3.28. "Follow path while recording" triggers the following drone routes after starting the recording. Finally, if "Generate RF after recording" is checked, after stopping the recording, RF generation starts with the provided RF configurations.



Figure 3.28. Segmented drone (ground truth) image example.

After the recording configuration is finished, pressing "Starts Recording" starts the recording, and "Stop Recording" ends the recording. After the recording finishes, the dataset is created under the recording folder set in the setting.json file. The dataset contains different folders and files based on configuration. An example of a dataset directory system is shown in Subsection 3.4.2.

3.5.10. Offline Data Generation

Because RF signal generation is a heavy process, and RF can be generated with different parameters for the same scenario we added offline RF signal generation under the "Offline Data Generation" tab shown in Figure 3.29. This RF signal generation function also uses the RF parameters.

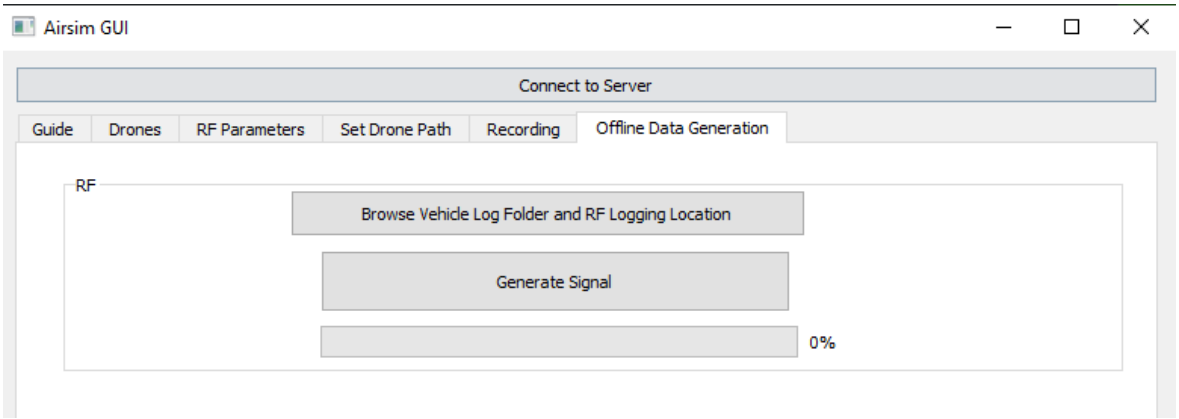


Figure 3.29. Airsim GUI "Offline Data Generation" tab.

Before starting the signal generation, the "vehicle_logs" folder must be selected from a previously created dataset. Then, a new directory needs to be created to save generated RF data and chosen as a save location. Finally, after pressing the "Generate Signal", data generation starts and the generated RF signals with the RF parameters are saved in a provided directory.

4. RF DATA MODEL AND GENERATION

RF generation is a special topic in this thesis. In the literature, there is no other example of generating RF data by using Airsim. We are using drone location records and processing them to create IQ signals collected by USRP in the simulation environment.

4.1. Pyphysim Library Usage

For RF data generation, we mainly used the pyphysim library in Python. In this library, there are different types of channel models. We focused on the single-user channel model.

4.2. RF Generation and Channel Model

4.2.1. Airsim GUI RF Parameters

We are taking different types of input from users to generate RF data. We described these input and valid types in the Subsection 3.5. These parameters are related to different parts of the model. For the OFDM part, we are taking three different inputs; FFT size, number of subcarriers, and cyclic prefix. In the channel, we implemented Jakes' model and it takes only the number of rays as an input. For the path loss model, we used a free space path loss model which takes the path loss coefficient and carrier frequency. In the other parameters part we are taking inputs for different parts of data generation steps.

In the right part of GUI, we mainly take json formatted data. We are using the tapped delay line (TDL) channel model which uses tap powers and tap delays. The length of them are equal and every index in these two arrays refer to the same tap information. Then USRP locations are acquired in Unreal Engine format. Users

can give different names to USRPs and can locate different numbers of them. Finally, Markov state-related inputs are taken in Markov state probabilities, Markov state metadata, and initial state sections. The Markov state is used in data transmission states in every iteration. These taken inputs are analyzed in Subsection 4.2.3.

4.2.2. RF Generation

The detailed RF generation steps are shown in Figure 4.1. Firstly, we calculate how many symbols are generated in a single iteration. This iteration length is fixed for every simulation and calculated based on user inputs. Then, based on the Markov state, RF signal output for active transmission or idle state is generated. The state architecture will be described in the following subsection in detail. If state 1 is selected, then an RF signal is generated and this signal is passed through the channel.

Path symbols are defined in signal generation using a random number generator. Then, using generated random symbols BPSK or QPSK modulation and OFDM modulation are applied in order. After these modulations and multiplying with transmitter gain, IQ data is prepared to pass through the channel. Our channel is a single-user channel and it contains Jakes' and TDL channel models. Because we support multiple USRPs in simulation after the single-user channel, other steps will be applied separately for every USRP. For every drone, USRP pair path loss is calculated individually, and then the phase difference is applied. After this step, the signals of other drones are summed together. After that, white noise and receiver gain are applied. Finally, the received IQ signal is generated for every USRP.

4.2.3. RF Data Generation Steps

We mentioned the steps for data generation in previous sections. This subsection will describe the RF data generation steps in detail. Described parts can be followed in Figure 4.1 with the related step numbers. In the diagram for every process, a corresponding number is placed.

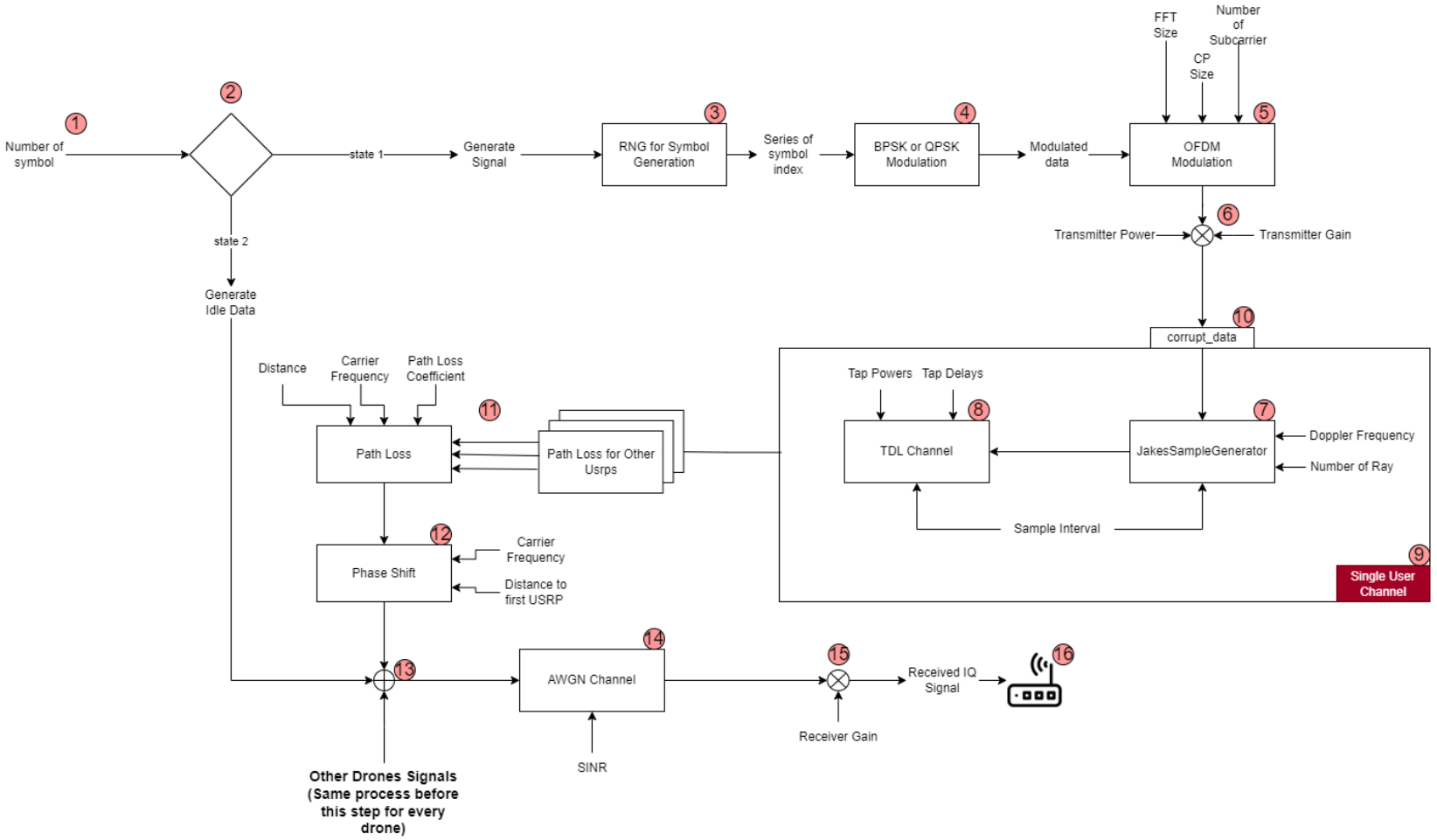


Figure 4.1. RF generation detailed diagram.

4.2.3.1. Calculate the Number of Symbols. In the simulation, we are generating RF data simulating the drones. For every drone, we have a Markov chain that decides, for the following time slots, to generate RF data for idle or busy states. For every state in the Markov chain, the same number of symbols are generated in a simulation based on given inputs. First of all, we are calculating this value with OFDM parameters.

We are using OFDM in the following steps. The OFDM block output length is affected by its parameters. And we are calling this output length 'RF Block Length'. RF Block Length parameter is calculates as

$$L_{rb} = n_{fs} + n_{cp}, \tag{4.1}$$

where L_{rb} is RF block length, n_{fs} is the FFT size, and n_{cp} is the cyclic prefix size.

We are taking how many RF blocks will be generated in a state as an input. Then we calculate how many symbols will be generated in a state as

$$L_{ss} = L_{rb} \times n_{rb}, \tag{4.2}$$

where, L_{ss} is the symbol length, and n_{rb} refers to how many RF blocks will be used in a state. We calculated the RF data lengths in a state with Equations (4.1) and (4.2). As described in Subsection 3.4.2 we create a log file for each drone. We are calculating the time difference between two consecutive logs to find how many states will run for RF data generation. We are using the state symbol block length that is calculated in Equation (4.2) and the number of IQ signals per second input parameter.

To ensure that the generated data length compatible with RF inputs, we are using three consecutive equations as

$$\begin{aligned} L_{sl} &= (n_{iqs} \times \Delta t_{log}) + r, && (4.3)\\ n_{sl} &= \left\lfloor \frac{L_{sl}}{L_{ss}} \right\rfloor, && (4.4)\\ r &= L_{sl} - (n_{sl} \times L_{sl}). && (4.5) \end{aligned}$$

In Equation (4.3), the number of symbols that need to be generated between two consecutive logs is given. n_{iqs} is the number of IQ signals per second parameter taken

from Airsim GUI. Δt_{log} is the time difference between two consecutive logs. r is the remainder and calculated in Equation (4.5). Because the symbol length of Markov states is fixed and calculated in Equation (4.2), the number of states could not fit the required symbols between two logs. We used a remainder to generate exactly the number of symbols based on the number of IQ signals per second. Equation (4.4) calculates the number of required states between two consecutive logs.

4.2.3.2. Markov Chain. We are deciding whether to send data from a drone by using a Markov chain (transmitting/busy state). Markov chain is created by connecting states. Every state has metadata and probabilities of moving to other states.

There is an example of a Markov chain in Figure 4.2. This Markov chain has two different states. These states refer to transmitting/busy and not-transmitting/idle states and used for signal generation. There are two probabilities for every state, and their values are written in Figure 4.2. Passing from state 0 to state 1 is 5% and staying in the same state is 95%. Passing from state 1 to state 0 is 10%, and staying in the same state is 90%. The sum of outgoing probabilities must be 1.

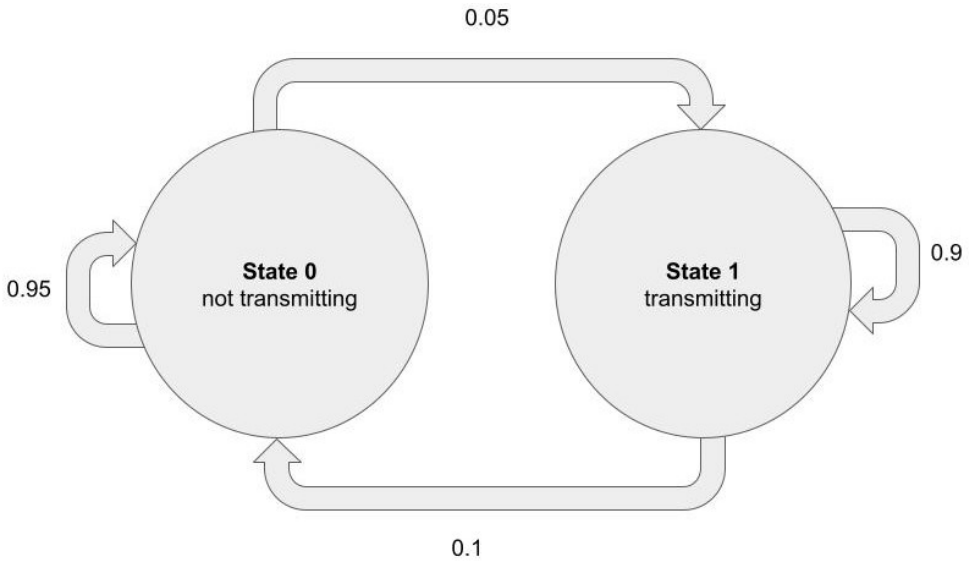


Figure 4.2. An example of Markov chain.

We are taking the Markov chain inputs from GUI by using two different JSON files, one of them for probabilities and the other one for metadata. We are taking transmitting information and if there is a transmission what is the modulation technique and the transmitting power as metadata.

4.2.3.3. RNG for Symbol Generation. For transmitting RF signals, we randomly generate symbols. Because we support two different modulation techniques in the next step, random values are changed based on it. For BPSK, generated symbols are 0 or 1, and for QPSK, they are 0, 1, 2, and 3. We are using Numpy's random function as a random number generator, and we are using uniform distribution.

4.2.3.4. Modulation. As described in the previous step we support two different modulation techniques: BPSK and QPSK. In this step, IQ data are generated for every symbol, and the power of generated signals is 1 in this step. For example Figure 4.3 shows the outputs of QPSK modulation. Four different red dots represent four different symbols. The x-axis represents I and the y-axis represents Q.

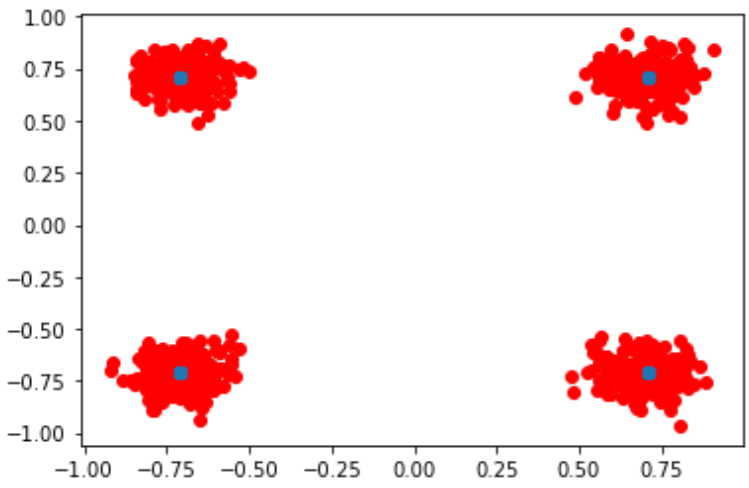


Figure 4.3. QPSK modulated data in IQ plane. Red points refer noisy signal and blue points refer to the original signal centers without noise.

4.2.3.5. OFDM (Digital) Modulation. We also use OFDM modulation in the transmitter. We used the standard OFDM processes and we can separate steps as in Figure 4.4. We are taking three parameters for OFDM: FFT size, cyclic prefix size, and number of subcarriers.

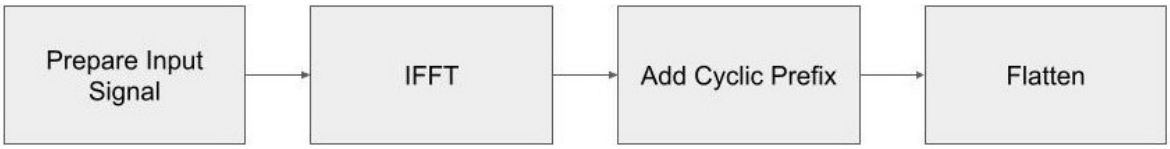


Figure 4.4. OFDM Diagram.

First, an input signal is prepared for OFDM. This process covers adding zero padding if it is required, and serial to parallel shifter. The input signal length for OFDM modulation needs to be equal to the number of subcarriers or integer multiples. If it is not zero padding is added at the end of signals. We ensure the signals do not need any zero padding when Airsim GUI is used. However, if someone uses our RF data generation functions rather than Airsim GUI zero padding can be applied. To calculate the length of zero padding, Equations (4.6) and (4.7) are performed. Then zero padding is concatenated into the input signal. After the zero padding process, the serial to the parallel shifter is applied. One-dimensional data is transformed into two-dimensional to modulate every row as one OFDM symbol. The shape will be "Number of OFDM symbols X Number of subcarriers". The number of OFDM symbols and the length of zero padding are calculated as

$$L_{ofdm} = \left\lceil \frac{L_{oi}}{n_{sc}} \right\rceil, \tag{4.6}$$

$$L_{zp} = (n_{sc} \times L_{ofdm}) - L_{oi}. \tag{4.7}$$

In these expressions, the generated OFDM symbol length and the required zero padding length are calculated respectively. L_{oi} is the input signal length of OFDM block and n_{sc} is the number of used subcarrier parameter.

For the next block input, which is IFFT, a new empty array is created. This array shape is "Number of OFDM symbols $\times$ FFT size". Because columns represent

subcarrier indexes, used subcarrier indexes are calculated and the prepared data is moved into this array. If the number of subcarriers is less than the FFT size, some columns will be filled with 0s. These columns represent DC subcarriers. Now the the data is ready for the IFFT block.

In IFFT block IFFT is performed with FFT size. Then the output is multiplied by the square root of the power that needs to be applied in the modulator. This total power is shared among useful samples and the cyclic prefix in one OFDM symbol. Therefore, the larger the cyclic prefix size the lower this power to account for energy loss due to sending the cyclic prefix. The power is calculated as

$$P_{ofdm} = \frac{n_{fs}^2}{n_{sc} + n_{cp}}. \tag{4.8}$$

After each IFFT output, the cyclic prefix is added as

$$\text{CP output} = \text{IFFT output}_n[:, -n_{cp}:] \,||\, \text{IFFT Output}_n[:, :], \tag{4.9}$$

where || stands for concatenation operation. To overcome intersymbol interference, a cyclic prefix extension is added to each frame.

At the end of the OFDM, data is flattened by transforming two-dimensional data to one-dimensional data. This process is called parallel to serial converter. The parallel time domain data is rearranged into serial frames.

4.2.3.6. Transmitter Power and Transmitter Gain. Once the OFDM block has been processed, transmitter power and transmitter gain need to be applied to the IQ signal. Transmitter power is taken from a user in a dB scale in Markov state inputs. Transmitter power is transformed into amplitude as

$$A_{\text{Tx}} = \sqrt{2 \times P_{\text{Tx}}}. \tag{4.10}$$

Also, we are taking transmitter gain information from a user as dB. Transmitter gain is transformed from dB to linear scale as

$$G_{\text{Tx}} = 10^{\frac{G_{\text{Tx}}(dB)}{10.0}}. \tag{4.11}$$

By using converted versions of transmitter power and transmitter gain, they can directly multiply them with OFDM output signal as

$$x_{ch}(t) = x_{ofdm}(t) \times G_{\mathrm{Tx}} \times A_{\mathrm{Tx}}, \tag{4.12}$$

where $x_{ch}(t)$ represents the signal that will pass through the channel and $x_{ofdm}(t)$ represents the output of the OFDM block.

4.2.3.7. Jakes' Channel Model. We developed the RF data generater model in a flexible manner and different channel models can be implemented easily for future works. We chose Jakes' channel model because it is based on the Rayleigh model and has a Doppler frequency parameter. It also contains different scatters that come from uniformly distributed angels and phases around a circle.

We took three parameters for Jakes' model: the Doppler frequency (Hertz), the sampling interval (second), and the number of rays. Only the Doppler frequency is not taken from a user. We are creating Jakes' sample generator object in every transmitting state to generate random values from scratch.

When Jakes sample generator object is created, first ψ_l and ϕ_l components are generated for every single ray by using the uniformly distributed random function as

$$\psi_l = 2 \times \pi \times \mathrm{random}(L,\, 1), \tag{4.13}$$

$$\phi_l = 2 \times \pi \times \mathrm{random}(L,\, 1). \tag{4.14}$$

In these expressions ψ_l and ϕ_l are L X 1 dimensional matrix. ψ_l is generated for random phases and ϕ_l is generated for random arrival time. The used random function refers to a uniformly distributed random function.

Then by using generated random phases and random arrival angels, the channel matrix is created as

$$h(t) = \frac{1}{\sqrt{L}} \sum_{l=0}^{L-1} \exp\left\{ j \left[2\pi F_d \cos(\phi_l) t + \psi_l \right] \right\}, \tag{4.15}$$

where L is the number of rays and F_d is Doppler frequency.

4.2.3.8. TDL Channel Profile. In addition to Jakes's model, we also use the tapped delay line (TDL) channel model. TDL channel model is used to model the multipath characteristic of a wireless communication channel by using given tap delays and powers. We are taking tap delays and powers from a user as two arrays. Tap delays are taken as nanosecond and tap powers are taken as dB.

Before using the tap delays and tap powers, they are discretized with the used sampling interval. During the discretizing process, every tap delay is calculated and mapped which indexes they are mapped. The mapped delay index is calculated as

$$n_{di} = \left\lfloor \frac{\tau}{T_s} \right\rfloor, \tag{4.16}$$

where τ is the tap delay and the T_s is the sampling interval.

If there is more than one tap that maps the same index, their powers are summed together in linear scale, and new TDL channel profile arrays' lengths are shrunk.

4.2.3.9. Single User Channel. In the 7th and 8th steps, Jakes' and TDL channels were created. These channels are merged under a single user channel to send signals in this channel. Jakes' sample generator shape also changed to apply Jakes' model to every TDL channel tap.

4.2.3.10. Corrupt Data. Corrupt data represents a function in a single-user channel designed to transmit a signal through the channel. Initially, channel samples are generated with the shape 'Number of TDL channel taps × Number of symbols,' and Jakes' sample generator fills this matrix. To apply the TDL channel, each row is multiplied by the square root of the corresponding tap power in linear scale. Now, our channel samples are prepared to undergo both Jakes' and TDL models simultaneously.

Before applying these channel samples, an empty output array is created to store the resulting signal, incorporating zero padding. Zero padding is employed due to the applied tap delays extending to the last symbol.

```
1: for i, d  in enumerate(tdl_channel_tap_indexes) do
2:     output_signal[d : d + number_of_symbols] += channel_samples[i]*input_signal
3: end for
```

Figure 4.5. Corrupt data algorithm.

Then every row in the channel sample is multiplied by the input signal and then added to the output array based on TDL channel delay. The process can be seen in Figure 4.5.

4.2.3.11. Path Loss. Up to this point, the steps were common for all simulated USRPs. But in this step, it is separated for all USRPs since different location result in different path loss, phase shift, and random noise. For path loss, we implemented a free space path loss model. Input parameters for the free space path loss model are path loss coefficient, carrier frequency, and distance. Path loss coefficient and carrier frequency are taken from a user at the beginning of a simulation and distance is calculated for every single log. The path loss is calculated as

$$PL(dB) = 10 \cdot n \cdot \log_{10}(d) + 10 \cdot n \cdot \log_{10}(f_c \times 1e6) + 10 \cdot n \cdot \left(\frac{4\pi}{c}\right). \tag{4.17}$$

4.2.3.12. Phase Shift. Because there is a distance between USRP, there is also a phase difference for the received signal. The phase difference is calculated as

$$\theta = \frac{2\pi \Delta \mathrm{d} \times f_c}{c}, \tag{4.18}$$

where Δd is the distance difference according to the first USRP and the c is the speed of light. By using calculated phase difference phase shifting is applied to the signal.

4.2.3.13. Sum Other Drones' RF Signals. Until this step, every drone generates its RF signal individually. At this point generated signals by drones are superposed.

4.2.3.14. Additive White Gaussian Noise. We are adding an AWGN channel to create a noisy channel when there is no signal generation from drones (idle channel). We are taking the AWGN SINR parameter in dB and then we convert it into the linear scale and apply it to the IQ signal.

4.2.3.15. Receiver Gain. In addition to transmitter gain, receiver gain is also taken from a user. And similar to Equation (4.11), receiver gain is converted into a linear scale. Then the signal is multiplied by it.

4.2.3.16. USRP. Adding the AWGN channel was the last process. After that, for every USRP a file is created, and the generated signal is written into these files in a binary format. Also RF generation parameters are written into another file as an object.

5. EXAMPLE USE CASES

We created a dataset using the implemented simulation environment and Airsim GUI for the example use cases. We decided on six different drone paths as different scenarios. Decided scenarios are shown in Figure 5.1. We put the table in three different locations for every scenario and collected data during day and night hours in the simulation environment. In the end, the simulation count became 36. We spawned ten birds into the environment to raise the difficulty of drone detection via image data. The created dataset occupies approximately 43 GB.

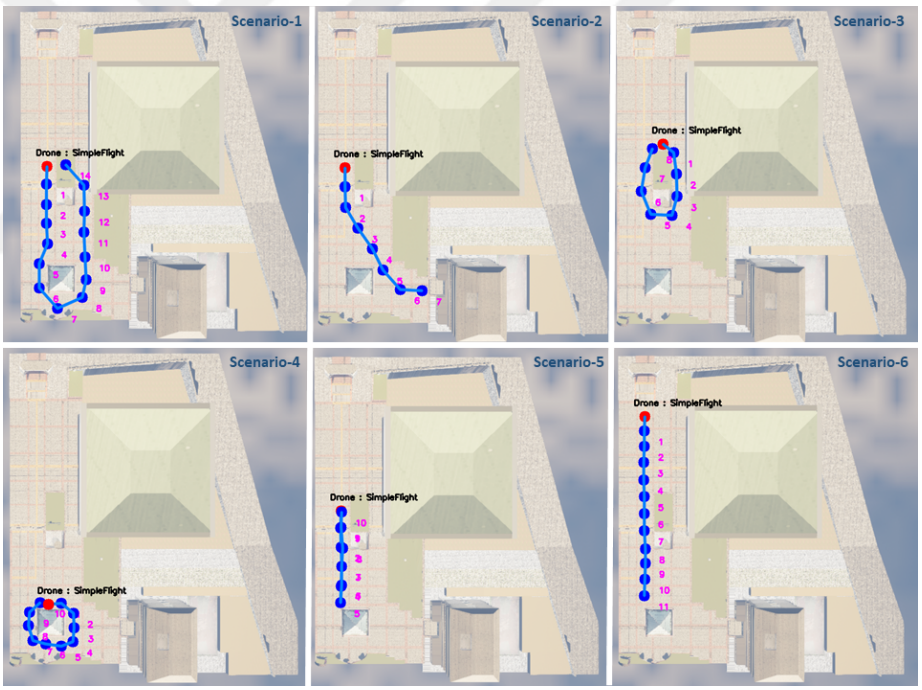


Figure 5.1. Dataset scenarios.

Because of the table configuration, there was one meter between cameras. We also put USRPs in table locations with a half meter between them. To summarise, we used two cameras, two USRPs, a drone, and ten birds for the simulation. Cameras views were recorded in the "Scene" camera type, and drones were segmented automatically using the recording feature in the Airsim GUI. Recorded images are full HD (1920X1080 pixels). RF parameters are used with default values except USRP locations.

We worked on three different problems for the use cases: drone detection via image data, drone detection via RF data, and drone distance estimation via RF data.

5.1. Drone Detection via Image Data

For the drone detection system, which uses just cameras, we used the YOLOv8 object detection model created by Ultralytics [32]. It is the state-of-the-art model for object detection and tracking, instance segmentation, image classification, and pose estimation. We used just an object detection model for this use case.

Ultralytics proposes five different-sized YOLOv8 models: nano(n), small(s), middle(m), large(l), and extra-large(x). Their main difference is the number of parameters used in the model. Comparison between these models for 640-pixel images trained on the COCO dataset [33] is shown in Table 5.1. As the table shows, increasing parameters increases the mean average precision(mAP) score and detection speed. Because speed is a crucial point for detection systems, we used the smallest model YOLOv8n. The bigger networks require more time to detect targets and more advanced hardware.

Table 5.1. YOLOv8 object detection model performance comparison (adapted from [34]).

Model	**mAP50-95**	**Speed (ms) (A100 TensorRT)**	**Params (M)**
YOLOv8n	37.3	0.99	3.2
YOLOv8s	44.9	1.20	11.2
YOLOv8m	50.2	1.83	25.9
YOLOv8l	52.9	2.39	43.7
YOLOv8x	53.9	3.53	68.2

Table 5.1 contains the mAP score, which stands for mean average precision. mAP score is one of the primary performance evaluation metrics for assessing an image

detection system. The numerical value following mAP indicates that the metric is computed by averaging precision values over a range of IoU thresholds from 0.5 to 0.95 with a step size of 0.05 as

$$mAP50-95 = \frac{1}{11}\sum_{t=0.5}^{0.95} mAPt. \tag{5.1}$$

The precision-recall curve corresponding to a specific IoU threshold determines the average precision. The formulas for precision and recall can be written as

$$\text{Precision} = \frac{\text{True Positives}}{\text{True Positives + False Positives}}, \tag{5.2}$$

$$\text{Recall} = \frac{\text{True Positives}}{\text{True Positives + False Negatives}}. \tag{5.3}$$

The average precision is the area under the precision-recall curve.

IoU is a measure of the overlap between the predicted bounding box and the ground truth bounding box. It is defined as the area of the intersection divided by the area of the union. It is expressed as

$$\text{IoU} = \frac{\text{Area of Intersection}}{\text{Area of Union}}. \tag{5.4}$$

YOLOv8 is divided into three parts: the backbone, neck, and head. These parts are used for feature extraction, multi-feature fusion, and prediction output. The architecture of the YOLOv8 network is illustrated in Figure 5.2.

The feature extraction network utilizes the C2f and SPPF modules to extract individual scale features from images. The C2f modules are inspired by CSP, and it reduces the amount of calculation needed for feature extraction [35, 36]. The SPPF module is used to extract features through three pooling operations, which improves the receptive field of the network. FPN (feature pyramid network) and PAN (path aggregation network) are used in the neck as a feature fusion method. YOLOv8 combined these features to improve feature fusion ability and solve the multi-scale target problem. The head part performs localization and classification tasks separately. Also, the head part uses an anchor-free approach rather than an anchor-based one used in earlier YOLO versions. Anchor-free object detection methods eliminate the need for

predefined anchor boxes, offering increased flexibility and adaptability to diverse object scales and aspect ratios in a more streamlined architecture.

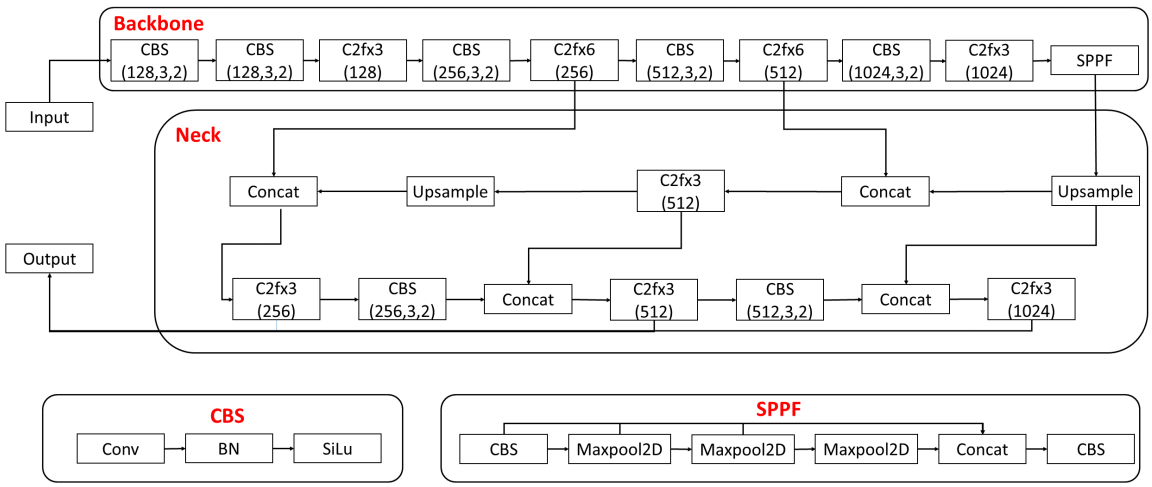


Figure 5.2. YOLOv8 network architecture (adapted from [36]). CBS components are composed of Convolution, Batch Normalization, and SiLu activation function. The SPPF is composed of three Maxpool tiers and two CBS components.

To train the model, labels of the dataset are prepared for the required format. The dataset has segmented images as ground truth, as shown in Figure 3.28. However, the YOLOv8 drone detection model accepts bounding box center coordinates and the height and width. The labels are prepared based on it. The dataset was divided into training validation and test datasets as 60%-20%-20%, respectively.

The hardware configuration used for the experiment is an 8GB Nvidia GTX 1070 graphic card, the deep learning framework PyTorch 1.13.1, Python version 3.8, CUDA version 11.0, and Windows 10 as the operating system. The YOLOv8n model is trained by the created dataset to detect only drones. The training process was run for 1 hour and 15 minutes, with the pre-trained model on the COCO dataset. We trained the model in a 1024x576 pixel image format. Because of that, for every image, preprocessing is applied for resizing the images. After 14 epochs, the mAP50 became 0.73839. The overall training metrics are shown in Table 5.2.

Table 5.2. YOLOv8n training process for drone detection via image dataset.

Epoch	**Precision**	**Recall**	**mAP50**	**mAP50-95**
1	0.76755	0.53322	0.55266	0.2614
2	0.61498	0.51329	0.53621	0.27792
3	0.74676	0.48494	0.54547	0.27878
4	0.80832	0.5604	0.61579	0.35433
5	0.68501	0.52492	0.5474	0.29289
6	0.80637	0.62261	0.67955	0.40068
7	0.85189	0.64014	0.68377	0.40726
8	0.82561	0.65275	0.72545	0.44725
9	0.84713	0.65615	0.70576	0.41265
10	0.91571	0.59552	0.71363	0.4251
11	0.87754	0.6711	0.73256	0.4228
12	0.85971	0.62292	0.707	0.41085
13	0.8656	0.63953	0.71531	0.44009
14	0.90008	0.65947	0.73839	0.45815

We calculated the model's speed using the prepared testing dataset for testing. On average, for preprocessing, 4.2ms is spent, and for inference, 12.8ms is used.

After training the model, we analyzed it for two cameras simultaneously. The main reasons for using two cameras are drones can be seen from only a single camera because of occlusion, and the model could not detect from a single camera. For this approach, we processed all the images through the model and analyzed every image frame captured at the same time. In the dataset, there are 7616 images; half (3808) were captured from the left camera, and the other half from the right camera. The drone can seen 2331 times from the left camera and 2155 from the right camera. When capturing a drone from only a single camera counted as there is a drone, a drone exists 2383 times in 3808 time units. So when a drone exists, the left camera can not detect 904 times, and the right camera can not detect 911 times. However, when trusting a

single camera, the detection failure is decreased to 798. The ROC curve for the drone detection with two cameras is shown in Figure 5.3, and confusion matrices can be seen in Table 5.3.

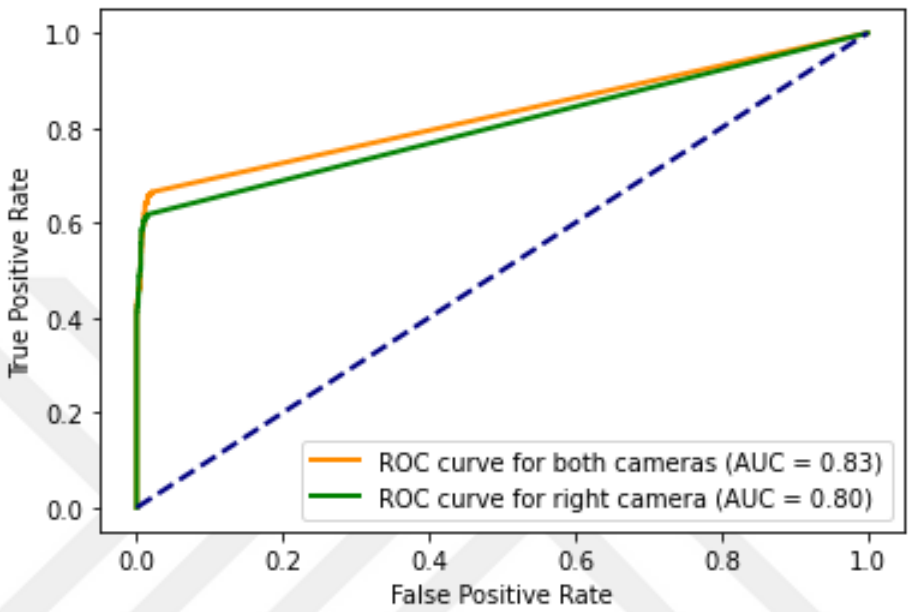


Figure 5.3. ROC curves for drone detection with two cameras and a single camera. The average confidence score is used as a threshold for detection with two cameras.

5.2. Drone Detection via RF Data

For the drone detection model via RF data, we used the same model in the detection via camera. However, we used a classification task rather than the object detection task. For classification, performance metrics are different than the detection. For classification, again, there are five different sizes. For different-sized YOLOv8 models for 244-pixel images, which are trained with ImageNet dataset [37], performance metrics are shown in Table 5.4.

The primary evaluation metric for classification tasks is accuracy. Top1 and top5 accuracies are used to compare models in the table. The top1 accuracy indicates how many times the network predicts the correct label with the highest probability. The top5 accuracy indicates the correct label how many times it appears in the network's

top 5 predicted classes. Because we used the model for binary classification, the top1 accuracy is used in our case.

Table 5.3. Confusion matrices of drone detection by using a single and two cameras. When a single camera can see the drone, it means positive. OR logic is used when both cameras are used.

Left Camera		**Actual**	
		Positives	**Negatives**
Predicted	**Positives**	1479	22
	Negatives	904	1403
Right Camera		**Actual**	
		Positives	**Negatives**
Predicted	**Positives**	1472	22
	Negatives	911	1403
Left and Right Camera		**Actual**	
		Positives	**Negatives**
Predicted	**Positives**	1585	31
	Negatives	798	1394

In our created dataset, there are only true cases, meaning a drone exists. For false cases, we generated Gaussian noise signals and we called it noise class. We split our generated signals into the 0.25-second sequences and labeled them. Because YOLOv8 is a model for images, we calculated the short-time Fourier transform(STFT) of the signals and saved them as an image. Because we used two USRP in the simulation environment, we merged two USRP signals' STFTs vertically. Then, generated images were divided into training validation and test datasets as 60%-20%-20%, respectively.

The used training environment is the same as the environment in section 5.1. We choose YOLOv8n-cls as a classifier with a 768x768 pixel image. After the first epoch, the accuracy metric did not change much, as seen in Table 5.5. The best accuracy is calculated in the third epoch, which is 0.92531.

We calculated the model's speed using the prepared testing dataset for testing. On average, 6.6 ms and 6.4 ms were spent on preprocessing and inference, respectively.

Table 5.4. YOLOv8 classification model performance comparison (adapted from [38]).

Model	**acc top1**	**acc top5**	**Speed (ms) (TensorRT)**	**Params (M)**
YOLOv8n-cls	66.6	87.0	0.31	2.7
YOLOv8s-cls	72.3	91.1	0.35	6.4
YOLOv8m-cls	76.4	93.2	0.62	17.0
YOLOv8l-cls	78.0	94.1	0.87	37.5
YOLOv8x-cls	78.4	94.3	1.01	57.4

Table 5.5. YOLOv8n-cls classification training process for drone detection via RF dataset.

Epoch	**acc top1**
1	0.92304
2	0.92485
3	0.92531
4	0.92531

5.3. Drone Distance Detection via RF Data

As a final use case, we created a custom convolutional neural network (CNN) to predict the distance between USRPs and the drone. For the USRP location, we chose the middle of the USRPs.

We trained two different models for 0.25 and 0.50 second sequences. We split our signals into 0.25 and 0.50 second sequences with distance as a label. For 0.50-second sequences, we start the sequences for every 0.25 second to create a similar number of data. We chose the average distance for sequences as a label. Because the sampling

frequency was 10^6, 0.25-second sequences contain 250,000 time points, and 0.50-second sequences contain 500,000 time points. For an input of the model, we split the I and Q signals for both USRPs and merge them vertically, as shown in Figure 5.4. So, we created 4x250000 and 4x500000 shaped arrays as input for our CNN models. The dataset was divided again into training, validation, and test datasets as 60%-20%-20%, respectively.

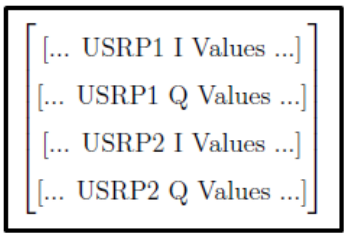
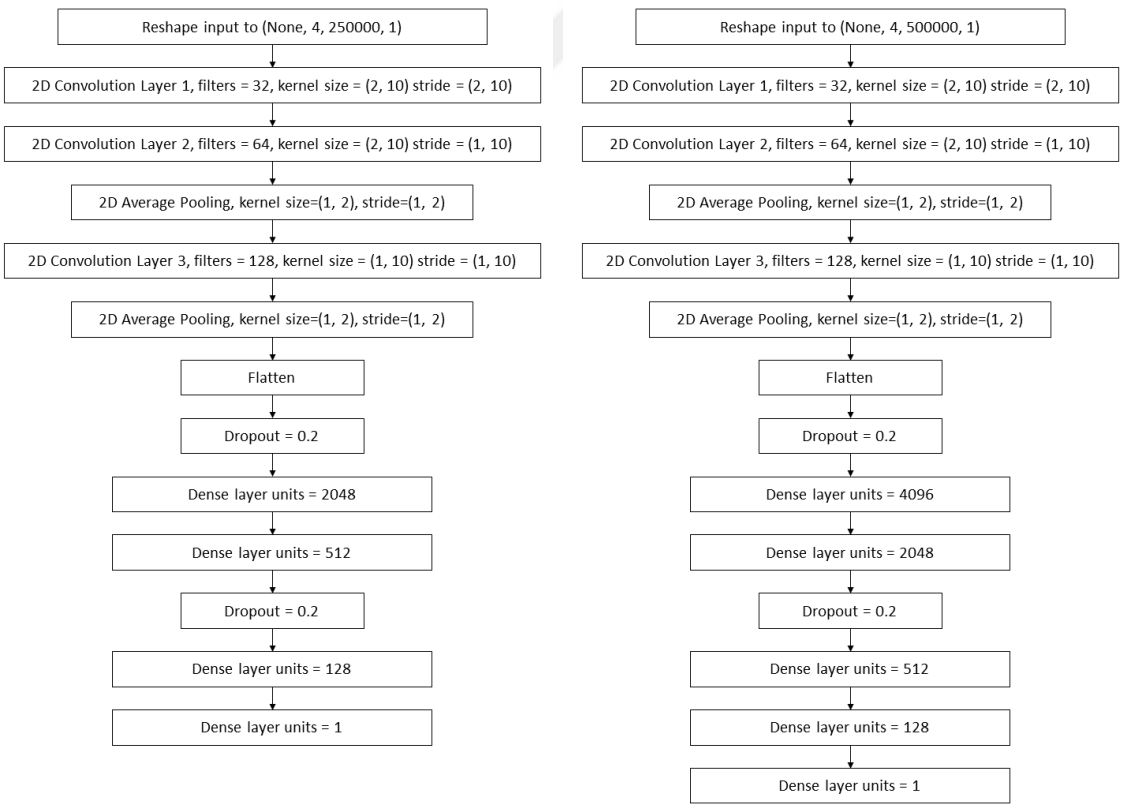


Figure 5.4. Input matrix format of drone distance detection model.



(a) (b)

Figure 5.5. Drone distance detection CNN model. (a) 0.25-second sequences, (b) 0.50-second sequences.

We propose the CNN architectures shown in Figure 5.5 for the drone distance detection system. Our CNN models has three 2D convolution layers and two 2D average pooling layers. To avoid overfitting two dropout layers with 20% probability added. Because the input shape for 0.50-second sequences is multiple times of 0.25-second sequences, we added one more dense layer to it.

Table 5.6. Drone distance detection statistics for the absolute difference and percentage error between ground truth and estimations. For statistics, we trimmed rows that in the min 10% and max 10% percentage error.

0.25 second frames (RMSE = 7.29) (Epoch 8)							
Absolute Difference (m)	**mean**	**std**	**min**	**25%**	**50%**	**75%**	**max**
	6.26	3.73	0.93	3.16	5.70	8.54	17.98
Percentage Error (%)	**mean**	**std**	**min**	**25%**	**50%**	**75%**	**max**
	14.82	7.73	3.33	8.20	14.10	20.22	34.23
0.50 second frames (RMSE = 6.07) (Epoch 10)							
Absolute Difference (m)	**mean**	**std**	**min**	**25%**	**50%**	**75%**	**max**
	5.32	2.93	0.64	2.94	4.65	7.16	14.96
Percentage Error (%)	**mean**	**std**	**min**	**25%**	**50%**	**75%**	**max**
	12.50	5.53	3.17	7.92	11.76	16.72	25.71

Training these models runs in the same hardware configurations with the hyper-parameters. Training these models runs in the same hardware configurations with the hyperparameters. The batch size is chosen as 10. Adam optimizer is used, and the learning rate is chosen as 0.001. Rectified linear units are used as an activation layer. The model for 0.25 second frames gave better results at the 8th epoch and 0.50 second frames at the 10th epoch in the test dataset. The model for 0.50 second frames gave better results than the model for 0.25 second. However, it requires 0.25 seconds more RF signal. The root mean square errors are 7.29 and 6.07 for 0.25 second and 0.50 second models, respectively. Also, absolute differences and percentage error statistics are shown in Table 5.6.

6. CONTRIBUTIONS AND FUTURE WORK

6.1. Summary of Contributions

The main contributions of this thesis can be summarized as:

- To overcome the complexity and budget problems of building drone detection systems for research, we proposed Airsim GUI, which works with Unreal Engine. It provides building drone detection systems in a simulation environment and generates both image and RF datasets.
- We implemented a drone RF data generation model in Airsim GUI. Different transmitter, receiver, and channel configurations can managed in Airsim GUI to generate specific datasets.
- An image and RF dataset is generated using the proposed dataset generation system. We used the generated dataset for drone detection via image dataset, drone detection via RF data, and drone distance detection via RF data.

6.2. Future Work

There are different methods to detect drones, like acoustic and radar. As a future work, we plan to add an acoustic dataset generation feature to the Airsim GUI.

In the RF data generation part, we implemented limited channel model types and transmission methods. Also, we need to separate line-of-sight and non-line-of-sight channel models in the system. We plan to work on these weaknesses in our system.

REFERENCES

1. Federal Aviation Administration, `https://www.faa.gov/node/54496`, accessed on January 06, 2024.

2. Defense Here, "Türkiye'deki Kayıtlı İHA/Dron Sayısı 68 Bin 426'ya Ulaştı", `https://www.defensehere.com/tr/turkiyedeki-kayitli-iha-dron-sayisi-68-bin-426ya-ulasti`, accessed on January 06, 2024.

3. Jiang, N., K. Wang, X. Peng, X. Yu, Q. Wang, J. Xing, G. Li, Q. Ye, J. Jiao and Z. Han, "Anti-UAV: A Large-scale Benchmark for Vision-based UAV Tracking", *IEEE Transactions on Multimedia*, Vol. 25, pp. 486–500, 2021.

4. Zhao, J., G. Wang, J. Li, L. Jin, N. Fan, M. Wang, X. Wang, T. Yong, Y. Deng, Y. Guo, S. Ge and G. Guo, "The 2nd Anti-UAV Workshop & Challenge: Methods and Results", ArXiv:2108.09909 [cs], 2021.

5. Zhao, J., J. Li, L. Jin, J. Chu, Z. Zhang, J. Wang, J. Xia, K. Wang, Y. Liu, S. Gulshad, J. Zhao, T. Xu, X. Zhu, S. Liu, Z. Zhu, G. Zhu, Z. Li, Z. Wang, B. Sun, Y. Guo, S. Satoh, J. Xing and J. S. Shengmei, "The 3rd Anti-UAV Workshop & Challenge: Methods and Results", ArXiv:2305.07290 [cs], 2023.

6. Coluccia, A., M. Ghenescu, T. Piatrik, G. De Cubber, A. Schumann and L. Sommer, "Drone-vs-Bird Detection Challenge at IEEE AVSS2017", *IEEE International Conference on Advanced Video and Signal Based Surveillance (AVSS)*, pp. 1–6, Lecce, Italy, 2017.

7. Coluccia, A., A. Fascista, A. Schumann, L. Sommer, M. Ghenescu, T. Piatrik, G. De Cubber, M. Nalamati, A. Kapoor, M. Saqib *et al.*, "Drone vs. Bird Detection Challenge at IEEE AVSS2019", *Proceedings of the 16th IEEE International Conference on Advanced Video and Signal Based Surveillance (AVSS)*, pp. 1–7,

Taipei, Taiwan, 2019.

8. Coluccia, A., A. Fascista, A. Schumann, L. Sommer, A. Dimou, D. Zarpalas, M. Méndez, D. de la Iglesia, I. González, J.-P. Mercier, G. Gagné, A. Mitra and S. Rajashekar, "Drone vs. Bird Detection: Deep Learning Algorithms and Results from a Grand Challenge", *Sensors*, Vol. 21, No. 8, p. 2824, 2021.

9. Coluccia, A., A. Fascista, A. Schumann, L. Sommer, A. Dimou, D. Zarpalas, F. C. Akyon, O. Eryuksel, K. A. Ozfuttu, S. O. Altinuc, F. Dadboud, V. Patel, V. Mehta, M. Bolic and I. Mantegh, "Drone-vs-Bird Detection Challenge at IEEE AVSS2021", *IEEE International Conference on Advanced Video and Signal Based Surveillance (AVSS)*, pp. 1–8, Washington, DC, USA, 2021.

10. Coluccia, A., A. Fascista, A. Schumann, L. Sommer, A. Dimou, D. Zarpalas, N. Sharma, M. Nalamati, O. Eryuksel, K. A. Ozfuttu, F. C. Akyon, K. Sahin, E. Buyukborekci, D. Cavusoglu, S. Altinuc, D. Xing, H. U. Unlu, N. Evangelious, A. Tzes, A. Nayak, M. Bouazizi, T. Ahmad, A. Gonçalves, B. Ragault, R. Jain, Y. Matsuo, H. Prendinger, E. Jajaga, V. Rushiti, B. Ramadani and D. Pavleski, "Drone-vs-Bird Detection Challenge at ICIAP 2021", *International Conference on Image Analysis and Processing*, Vol. 13374, pp. 410–421, Lecce, Italy, 2022.

11. Li, J., J. Murray, D. Ismaili, K. Schindler and C. Albl, "Reconstruction of 3D Flight Trajectories from Ad-Hoc Camera Networks", *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 1621–1628, Las Vegas, NV, USA, 2020.

12. Allahham, M. S., M. F. Al-Sa'd, A. Al-Ali, A. Mohamed, T. Khattab and A. Erbad, "DroneRF Dataset: A Dataset of Drones for RF-Based Detection, Classification and Identification", *Data in Brief*, Vol. 26, p. 104313, 2019.

13. Swinney, C. J. and J. C. Woods, "RF Detection and Classification of Unmanned Aerial Vehicles in Environments with Wireless Interference", *International Con-*

ference on Unmanned Aircraft Systems (ICUAS), pp. 1494–1498, Athens, Greece, 2021.

14. Glüge, S., M. Nyfeler, N. Ramagnano, C. Horn and C. Schüpbach, "Robust Drone Detection and Classification from Radio Frequency Signals Using Convolutional Neural Networks", *Proceedings of the 15th International Joint Conference on Computational Intelligence - NCTA*, pp. 496–504, Rome, Italy, 2023.

15. Medaiyese, O. O., M. Ezuma, A. P. Lauf and A. A. Adeniran, "Hierarchical Learning Framework for UAV Detection and Identification", *IEEE Journal of Radio Frequency Identification*, Vol. 6, pp. 176–188, 2022.

16. Al-Sa'd, M. F., A. Al-Ali, A. Mohamed, T. Khattab and A. Erbad, "RF-Based Drone Detection and Identification Using Deep Learning Approaches: An Initiative Towards a Large Open Source Drone Database", *Future Generation Computer Systems*, Vol. 100, pp. 86–97, 2019.

17. Al-Emadi, S. and F. Al-Senaid, "Drone Detection Approach Based on Radio-Frequency Using Convolutional Neural Network", *IEEE International Conference on Informatics, IoT, and Enabling Technologies (ICIoT)*, pp. 29–34, Doha, Qatar, 2020.

18. Allahham, M. S., T. Khattab and A. Mohamed, "Deep Learning for RF-Based Drone Detection and Identification: A Multi-Channel 1-D Convolutional Neural Networks Approach", *IEEE International Conference on Informatics, IoT, and Enabling Technologies (ICIoT)*, pp. 112–117, Doha, Qatar, 2020.

19. Medaiyese, O. O., A. Syed and A. P. Lauf, "Machine Learning Framework for RF-Based Drone Detection and Identification System", *2nd International Conference On Smart Cities, Automation & Intelligent Computing Systems (ICON-SONICS)*, pp. 58–64, Tangerang, Indonesia, 2021.

20. Nemer, I., T. Sheltami, I. Ahmad, A. U.-H. Yasar and M. A. R. Abdeen, "RF-Based UAV Detection and Identification Using Hierarchical Learning Approach", *Sensors*, Vol. 21, No. 6, p. 1947, 2021.

21. Zhao, Z., Q. Du, X. Yao, L. Lu and S. Zhang, "A Two-Dimensional Deep Network for RF-based Drone Detection and Identification Towards Secure Coverage Extension", *IEEE 98th Vehicular Technology Conference*, pp. 1–5, Hong Kong, 2023.

22. Nguyen, P., T. Kim, J. Miao, D. Hesselius, E. Kenneally, D. Massey, E. Frew, R. Han and T. Vu, "Towards RF-based Localization of a Drone and Its Controller", *Proceedings of the 5th Workshop on Micro Aerial Vehicle Networks, Systems, and Applications*, pp. 21–26, New York, NY, United States, 2019.

23. Unlu, E., E. Zenou, N. Riviere and P.-E. Dupouy, "Deep Learning-Based Strategies for the Detection and Tracking of Drones Using Several Cameras", *IPSJ Transactions on Computer Vision and Applications*, Vol. 11, No. 1, pp. 1–13, 2019.

24. Singha, S. and B. Aydin, "Automated Drone Detection Using YOLOv4", *Drones*, Vol. 5, No. 3, p. 95, 2021.

25. Zhai, X., Z. Huang, T. Li, H. Liu and S. Wang, "YOLO-Drone: An Optimized YOLOv8 Network for Tiny UAV Object Detection", *Electronics*, Vol. 12, No. 17, p. 3664, 2023.

26. Seidaliyeva, U., D. Akhmetov, L. Ilipbayeva and E. T. Matson, "Real-Time and Accurate Drone Detection in a Video with a Static Background", *Sensors*, Vol. 20, No. 14, p. 3856, 2020.

27. Carrio, A., S. Vemprala, A. Ripoll, S. Saripalli and P. Campoy, "Drone Detection Using Depth Maps", *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 1034–1037, Madrid, Spain, 2018.

28. Carrio, A., J. Tordesillas, S. Vemprala, S. Saripalli, P. Campoy and J. P. How, "Onboard Detection and Localization of Drones Using Depth Maps", *IEEE Access*, Vol. 8, pp. 30480–30490, 2020.

29. Akyon, F. C., O. Eryuksel, K. A. Ozfuttu and S. O. Altinuc, "Track Boosting and Synthetic Data Aided Drone Detection", *IEEE International Conference on Advanced Video and Signal Based Surveillance (AVSS)*, pp. 1–5, Washington, DC, USA, 2021.

30. Diken, "Boğaziçi Üniversitesi Aptullah Kuran Kütüphanesi yıkılıyor", `https://www.diken.com.tr/bogazici-universitesi-aptullah-kuran-kutuphanesi-yikiliyor/`, accessed on January 06, 2024.

31. CoquiGames, "[FREE] Simple Birds Blueprint - Easily Add Birds to Your Projects!", `https://forums.unrealengine.com/t/free-simple-birds-blueprint-easily-add-birds-to-your-projects/120364`, accessed on January 06, 2024.

32. Jocher, G., A. Chaurasia and J. Qiu, "Ultralytics YOLOv8", 2023, `https://github.com/ultralytics/ultralytics`, accessed on January 06, 2024.

33. Lin, T.-Y., M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár and C. L. Zitnick, "Microsoft COCO: Common Objects in Context", *European Conference on Computer Vision*, pp. 740–755, Springer, Zurich, Switzerland, 2014.

34. Ultralytics, "Object Detection", 2023, `https://docs.ultralytics.com/tasks/detect/`, accessed on January 06, 2024.

35. Wang, C.-Y., H.-Y. Mark Liao, Y.-H. Wu, P.-Y. Chen, J.-W. Hsieh and I.-H. Yeh, "CSPNet: A New Backbone that can Enhance Learning Capability of CNN", *IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, pp. 1571–1580, Seattle, WA, USA, 2020.

36. Guo, J., H. Lou, H. Chen, H. Liu, J. Gu, L. Bi and X. Duan, "A New Detection Algorithm for Alien Intrusion on Highway", *Scientific Reports*, Vol. 13, No. 1, p. 10667, 2023.

37. Deng, J., W. Dong, R. Socher, L.-J. Li, K. Li and L. Fei-Fei, "Imagenet: A Large-Scale Hierarchical Image Database", *IEEE Conference on Computer Vision and Pattern Recognition*, pp. 248–255, Miami, FL, USA, 2009.

38. Ultralytics, "Image Classification", 2023, `https://docs.ultralytics.com/tasks/classify/`, accessed on January 06, 2024.