National Centre for Radio Astrophysics

Student Training Program 2024 Report

ANTENNA POINTING AND CONTROL SYSTEM DEVELOPMENT FOR RFI

(Arduino-Based Control System for Precise Antenna Positioning)

June – August, 2024

By

Madhav Santosh Hadge

Under Supervision of

Dr. Shubhendu Joardar & Mr. Yogesh Gaikwad

GIANT METERWAVE RADIO TELESCOPE
NATIONAL CENTER FOR RADIO ASTROPHYSICS
TATA INSTITUTE OF FUNDAMENTAL RESEARCH

Narayangaon, Tal-Junnar, Dist-Pune

Acknowledgement

I would like to express my profound gratitude to Dr. Shubhendu Joardar and Mr. Yogesh Gaikwad for giving me the opportunity to work on the ANTENNA POINTING AND CONTROL SYSTEM DEVELOPMENT FOR RFI project. Their guidance and support have been valuable throughout this journey.

I am deeply grateful to the NCRA Director, Prof. Y. Gupta, the GMRT Dean, Prof. Ishwar Chandra, and the Telemetry and Operation Group Coordinator. I would also like to extend my heartfelt thanks to the NCRA and GMRT staff, including Mrs. Deshmukh, for their assistance and support during my project.

Working with GMRT has been a truly wonderful experience. I had the opportunity to learn about GMRT’s work culture and interact with many experienced professionals. This interaction enriched my understanding and provided me with a broader perspective on the intricacies of antenna systems and their applications in astronomy. The insights I gained into the antenna pointing and control system for the RFI antenna were particularly valuable, allowing me to deepen my knowledge of the antenna pointing mechanism and the practical use of the antenna system.

This experience has significantly enhanced my technical skills and my understanding of real-world applications of the knowledge I have gained in the field of astronomy. I have developed a more comprehensive understanding of the complexities involved in antenna control systems and the importance of precision in the detection of RF signals. The hands-on experience with the antenna pointing mechanism has been instrumental in improving my problem-solving skills and my ability to apply theoretical knowledge to practical challenges.

In conclusion, I am immensely thankful for the opportunity to work on this project. The support and guidance from everyone at GMRT and NCRA have been crucial to my learning and growth. I look forward to applying the skills and knowledge I have gained to future projects and continuing my journey in the field of astronomy.

Madhav Santosh Hadge

Abstract

The primary objective of this project is to achieve precision and total control over an antenna system, with a focus on antenna pointing to detect RF signals effectively. This involves the integration of an Arduino microcontroller as the primary controller and Python for providing user inputs. The Arduino will receive commands specifying the desired angle and duration for which the antenna should move, and it will execute these commands with high precision. The precision of these movements is crucial for accurate signal detection, as the antenna must be positioned accurately to capture the desired RF signals.

The system's architecture involves a seamless interaction between hardware and software components. The Arduino is interfaced with motor drivers that control the motors responsible for adjusting the antenna's position. The motors must be capable of fine movements to ensure the antenna can be directed with high accuracy. On the software side, a Python-based user interface allows users to input the desired angle and time parameters. These inputs are then communicated to the Arduino, which processes the commands and drives the motors accordingly. The Python interface not only provides a user-friendly means of entering commands but also handles any necessary calculations and data processing to ensure the inputs are translated into precise movements.

A critical aspect of the project is the feedback mechanism to ensure precision. Sensors such as encoders or potentiometers may be employed to provide real-time feedback on the antenna's position, allowing the system to make necessary adjustments and corrections. This feedback loop is essential for maintaining the desired accuracy and ensuring the antenna remains pointed correctly throughout the specified duration.

Additionally, the system must be robust and reliable, capable of operating in various environmental conditions without significant degradation in performance. The integration of error-handling mechanisms and calibration routines will be crucial to achieving consistent and reliable operation. The overall design must also consider power management to ensure the system can operate for extended periods without interruption.

This project aims to develop a highly precise and controllable antenna system for detecting RF signals. By leveraging the capabilities of an Arduino controller and a Python-based input interface, the system will allow for precise adjustments to the antenna's position based on user-defined parameters. The success of this project hinges on the seamless integration of hardware and software components, robust feedback mechanisms, and meticulous attention to precision and reliability in the system's design and operation.

INDEX

Chapter 1
Introduction to GMRT (Giant Metrewave Radio Telescope (GMRT)

The Giant Metrewave Radio Telescope (GMRT) is a premier facility for radio astronomical research located near Pune, India. It is one of the largest and most sensitive radio telescopes in the world, operating in the meter wavelength range.

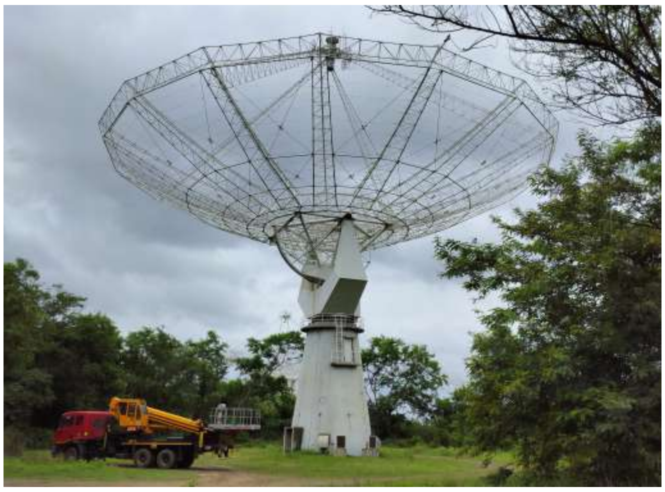


Fig. GMRT Antenna

- **Location:** Near Pune, Maharashtra, India.
- **Inception:** Commissioned in 2002.
- **Primary Purpose:** Observing celestial objects and phenomena at meter wavelengths.
- **Number of Antennas:** 30.
- **Type:** Parabolic dish antennas.
- **Diameter:** Each antenna is 45 meters.
- **Configuration:** Distributed in a Y-shaped array, similar to the Very Large Array (VLA) in the USA, with baselines (distances between antennas) ranging up to 25 km.
- **Frequency Range:** Operates in the frequency range of approximately 150 MHz to 1420 MHz.

1.1 Design and Construction:

The GMRT was designed and constructed by the National Centre for Radio Astrophysics (NCRA) of the Tata Institute of Fundamental Research (TIFR). The design leveraged local resources and expertise, making it a cost-effective project.

Unique Features: The GMRT is unique due to its large number of antennas and the wide range of frequencies it can observe. It is one of the few radio telescopes capable of operating at such low frequencies, which are crucial for studying various astronomical phenomena.

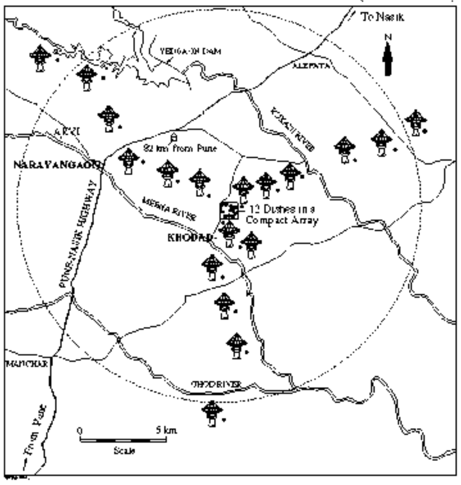
1.2 Antennas and Array Configuration:-

- **Antennas Design:** Each antenna is a fully steerable parabolic dish, 45 meters in diameter. The antennas are designed to be highly sensitive and can be pointed in different directions to observe different parts of the sky.

- **Materials:** The dishes are made of wire mesh, which is lightweight yet effective for reflecting radio waves at the GMRT's operating frequencies.

1.3 Array Configuration:-

LOCATIONS OF GMRT ANTENNAS (30 dishes)



- **Y-Shape Layout:** The 30 antennas are arranged in a Y-shaped configuration, with each arm of the Y being 14 km long. This layout provides good coverage of the sky and helps in achieving high-resolution imaging.
- **Central Cluster:** Twelve antennas are located in a compact central array within a 1 km region, enhancing sensitivity for observations of small-scale structures.

1.4 Operations and Technology:-

➢ **Interferometry and Aperture Synthesis:**

- **Interferometry:** The GMRT uses radio interferometry, where signals from pairs of antennas are combined to simulate a much larger telescope. This method enhances the resolution and sensitivity of the observations.
- **Aperture Synthesis:** By observing a source for an extended period and combining data from different antenna pairs, the GMRT can create high-resolution images of celestial objects, a technique known as aperture synthesis.

➢ **Receivers and Signal Processing:**

- **Receivers:** Each antenna is equipped with receivers that amplify and convert the radio signals into a form that can be processed digitally.
- **Signal Correlation:** The signals from all the antennas are brought to a central processing facility, where they are correlated. The correlator is a supercomputer that processes the vast amount of data to produce images and spectra.

➢ **Data Analysis:-**

- **Software:** The GMRT employs advanced software for data calibration, imaging, and analysis. Tools like AIPS (Astronomical Image Processing System) and CASA (Common Astronomy Software Applications) are commonly used.

- **Research Output:** The processed data enable scientists to study a wide range of phenomena, from the early universe and galaxy evolution to the interstellar medium and transient events like pulsars and fast radio bursts (FRBs).

1.5 Scientific Contributions:-

- **Cosmology and Galaxy Formation:** The GMRT has been instrumental in studying the large-scale structure of the universe and the formation and evolution of galaxies.

- **Pulsars and Neutron Stars:** The high sensitivity of the GMRT makes it ideal for discovering and studying pulsars, which are rapidly rotating neutron stars that emit regular radio pulses.
- **Interstellar Medium:** Observations of the 21 cm hydrogen line provide valuable insights into the distribution and properties of neutral hydrogen in the Milky Way and other galaxies.
- **Transient Phenomena:** The GMRT's wide field of view and sensitivity allow it to detect transient events like fast radio bursts (FRBs), which are brief but powerful radio pulses of unknown origin.

1.6 Upgrades and Future Prospects:-

- **GMRT Upgrade (uGMRT):** The GMRT has undergone a significant upgrade to enhance its capabilities. The uGMRT project has improved sensitivity, extended the frequency coverage, and upgraded the receivers and digital back-end systems.
- **International Collaborations:** The GMRT collaborates with various international observatories and participates in global scientific projects, contributing to the worldwide effort in advancing our understanding of the universe.

The GMRT remains a vital facility for the global astronomy community, continuously contributing to groundbreaking discoveries and advancing our knowledge of the cosmos.

CHAPTER 2
EXISITING SYSTEM

2.1 EXISTING SYSTEM MODEL:-

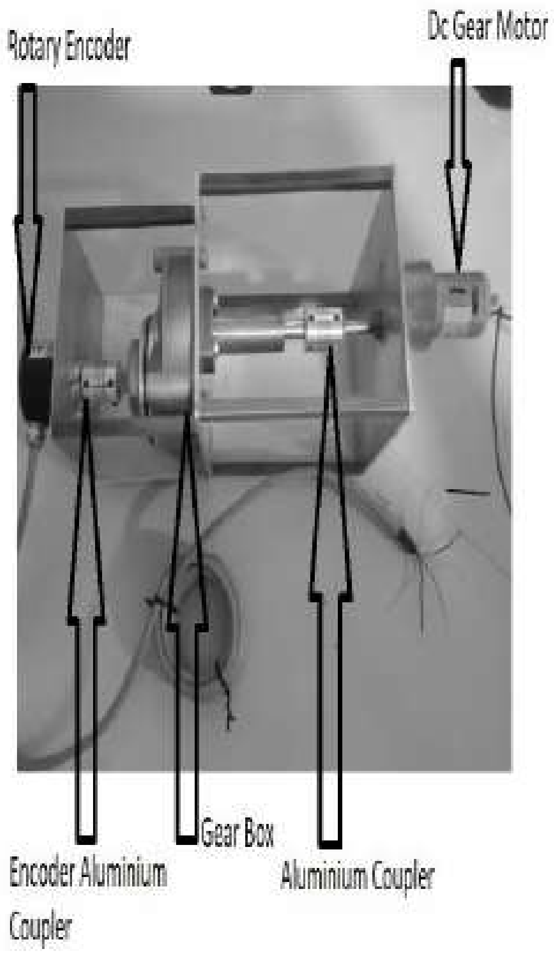
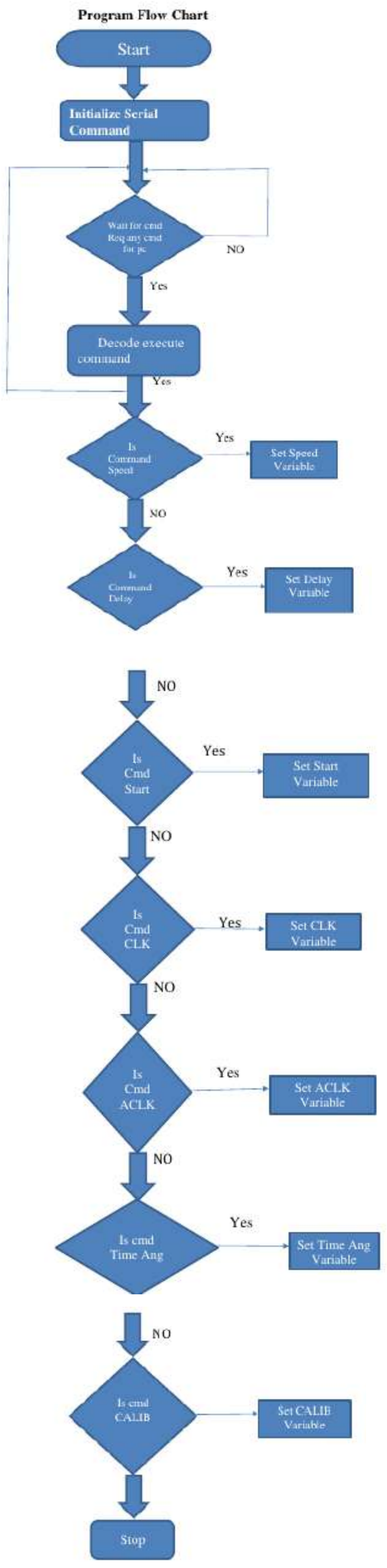


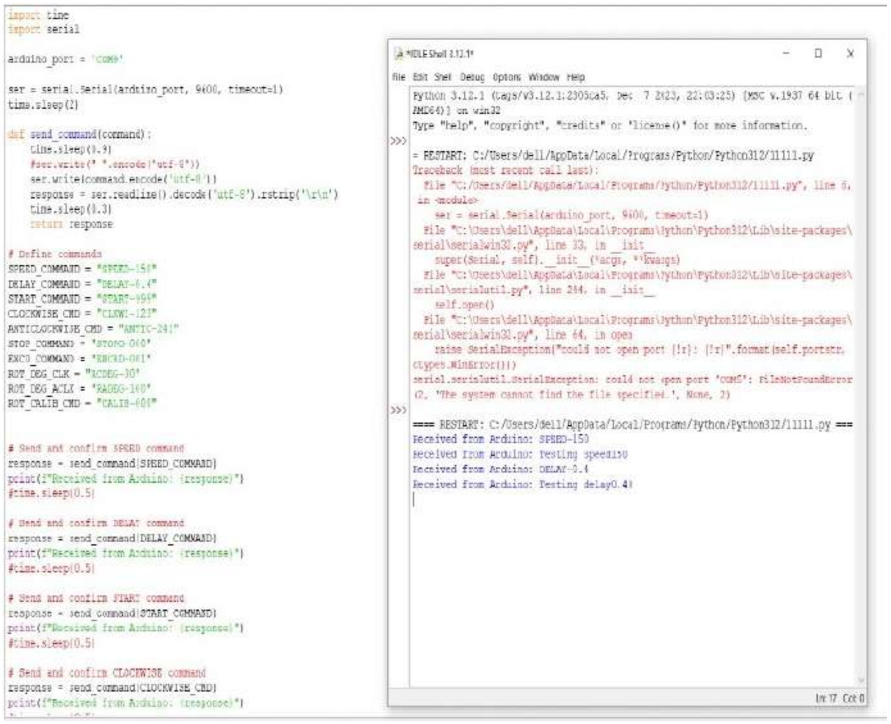
Fig. Refernece model

The above figure shows the Output of the prevoiusly made model. There was a trail based model which had same configuration as the real model. After selecting the direction from the arduino serial monitor. And using the serial monitor to give the direction of the rotation. This cyclic process is continued throughout the observation.

2.2 Flowchart for System Operation

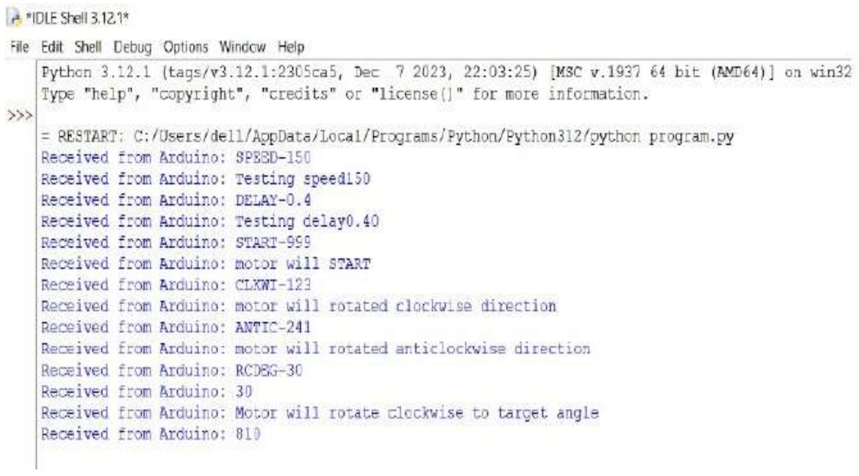


2.3 OUTPUT OF THE PREVIOUSLY EXISITNG PROGRAM:-



```
import time
import serial

arduino_port = 'COM5'

ser = serial.Serial(arduino_port, 9600, timeout=1)
time.sleep(2)

def send_command(command):
    time.sleep(0.3)
    #ser.write(" ".encode('utf-8'))
    ser.write(command.encode('utf-8'))
    response = ser.readline().decode('utf-8').rstrip('\r\n')
    time.sleep(0.3)
    return response

# Define commands
SPEED_COMMAND = "SPEED-150"
DELAY_COMMAND = "DELAY-0.4"
START_COMMAND = "START-999"
CLOCKWISE_CMD = "CLKWI-123"
ANTICLOCKWISE_CMD = "ANTIC-241"
STOP_COMMAND = "STOPO-000"
ENCO_COMMAND = "ENCRD-001"
ROT_DEG_CLK = "RCDEG-30"
ROT_DEG_ACLK = "RADEG-160"
ROT_CALIB_CMD = "CALIB-000"


# Send and confirm SPEED command
response = send_command(SPEED_COMMAND)
print(f"Received from Arduino: {response}")
#time.sleep(0.5)

# Send and confirm DELAY command
response = send_command(DELAY_COMMAND)
print(f"Received from Arduino: {response}")
#time.sleep(0.5)

# Send and confirm START command
response = send_command(START_COMMAND)
print(f"Received from Arduino: {response}")
#time.sleep(0.5)

# Send and confirm CLOCKWISE command
response = send_command(CLOCKWISE_CMD)
print(f"Received from Arduino: {response}")
```

```
Python 3.12.1 (tags/v3.12.1:2305ca5, Dec  7 2023, 22:03:25) [MSC v.1937 64 bit (
AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
= RESTART: C:/Users/dell/AppData/Local/Programs/Python/Python312/11111.py
Traceback (most recent call last):
  File "C:/Users/dell/AppData/Local/Programs/Python/Python312/11111.py", line 6,
 in <module>
    ser = serial.Serial(arduino_port, 9600, timeout=1)
  File "C:\Users\dell\AppData\Local\Programs\Python\Python312\Lib\site-packages\
serial\serialwin32.py", line 33, in __init__
    super(Serial, self).__init__(*args, **kwargs)
  File "C:\Users\dell\AppData\Local\Programs\Python\Python312\Lib\site-packages\
serial\serialutil.py", line 244, in __init__
    self.open()
  File "C:\Users\dell\AppData\Local\Programs\Python\Python312\Lib\site-packages\
serial\serialwin32.py", line 64, in open
    raise SerialException("could not open port {!r}: {!r}".format(self.portstr,
ctypes.WinError()))
serial.serialutil.SerialException: could not open port 'COM5': FileNotFoundError
(2, 'The system cannot find the file specified.', None, 2)
>>>
==== RESTART: C:/Users/dell/AppData/Local/Programs/Python/Python312/11111.py ===
Received from Arduino: SPEED-150
Received from Arduino: Testing speed150
Received from Arduino: DELAY-0.4
Received from Arduino: Testing delay0.40
```



```
Python 3.12.1 (tags/v3.12.1:2305ca5, Dec  7 2023, 22:03:25) [MSC v.1937 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
= RESTART: C:/Users/dell/AppData/Local/Programs/Python/Python312/python program.py
Received from Arduino: SPEED-150
Received from Arduino: Testing speed150
Received from Arduino: DELAY-0.4
Received from Arduino: Testing delay0.40
Received from Arduino: START-999
Received from Arduino: motor will START
Received from Arduino: CLKWI-123
Received from Arduino: motor will rotated clockwise direction
Received from Arduino: ANTIC-241
Received from Arduino: motor will rotated anticlockwise direction
Received from Arduino: RCDEG-30
Received from Arduino: 30
Received from Arduino: Motor will rotate clockwise to target angle
Received from Arduino: 810
```

The previous system required users to input values through the serial monitor, which was inefficient and cumbersome. The rotation and direction values were hardcoded into the program, leading to a rigid operation where the motor continuously cycled through forward and backward rotations without the ability to pause or adjust.

Additionally, the system's reliance on serial communication created significant delays. Values had to be transmitted to the Arduino one at a time, necessitating pauses in the program to ensure proper reception and processing. This sequential transmission slowed down the system, compromising its responsiveness, especially in applications where timing and precision were critical.

Moreover, the lack of flexibility was a major drawback. Once the motor started its operation, there was no way to interrupt or modify the cycle in real-time, making it difficult to adapt to changing conditions or user needs. Any adjustments required reprogramming, which was impractical for most users.

These limitations underscored the need for a more dynamic and user-friendly approach to motor control, one that would allow for real-time adjustments, minimize delays, and provide greater control over the system's operation.

CHAPTER 3
PROBLEMS WITH EXISTING SYSTEM

The precision of an antenna's movement is critical for detecting RF signals accurately. Any error in the antenna program can lead to physical damage and operational failures. To achieve the necessary precision, all components of the antenna system, from the motors to the GUI, must function correctly and in harmony. Developing a code that synchronizes the antenna with a Python GUI is crucial, yet numerous challenges have been encountered.

Previously, the antenna system was controlled using the Arduino IDE, which proved to be unreliable. The code was tested using a reference model, which did not accurately replicate the real-world problems encountered during implementation. Consequently, several issues arose:

1. Inaccurate Rotation of the Antenna : The antenna failed to achieve a full 360-degree azimuth rotation.

2. Lack of GUI for Motor Control : Previously, there was no graphical user interface (GUI) to control the motor, complicating the control process.

3. No Physical Antenna Implementation : The previous phase of the project did not include an actual antenna, which limited the scope of testing and validation.

4. Motor Control Issues : When the code was implemented, the motor did not stop at the desired angle, indicating a problem with precision and control.

5. Platform-Related Errors : The existing system, which ran on Windows, frequently encountered errors during file execution, necessitating a switch to Debian 12 with KDE Plasma desktop for better stability.

6. Serial Communication Problems : There were interruptions and errors in the serial communication between the Arduino and the Python machine, disrupting the synchronization and control processes.

Given these challenges, it is essential to redesign the antenna system to have simpler circuitry to reduce potential complications. This includes developing a reliable and precise control system integrated with a Python-based GUI, ensuring smooth communication and control of the antenna for accurate RF signal detection.

CHAPTER 4
SYSTEM REQUIREMENTS

4.0. Software Specification:-

- Operating System
- Programming Language and Software

4.1 Operating System:-

Debian 12, codenamed "Bookworm," represents a significant milestone in the evolution of the Debian GNU/Linux operating system. Released in June 2023, Debian 12 builds on the solid foundation of its predecessors, offering enhanced performance, improved security, and a rich set of features that cater to a wide range of use cases, from desktop environments to server deployments.

One of the most notable aspects of Debian 12 is its comprehensive update of software packages. This release includes over 11,000 new packages, bringing the total to around 64,000. Users benefit from updated versions of key applications and tools, such as GNOME 43, KDE Plasma 5.27, and the latest versions of popular programming languages and libraries. These updates ensure that users have access to the latest features and security enhancements, making Debian 12 a robust and versatile platform for both personal and professional use.

Debian 12 also emphasizes security and stability, which have always been hallmarks of the Debian project. This release incorporates numerous security improvements, including enhanced support for Secure Boot, which ensures that the system boots only trusted software. Additionally, the Debian Security Team continues to provide timely updates and patches for vulnerabilities, ensuring that Debian 12 remains a secure choice for users who require a dependable operating system for their critical workloads.

Another key feature of Debian 12 is its commitment to accessibility and user experience. The installer has been refined to be more user-friendly, with better hardware detection and improved support for a wide range of devices. The default desktop environment, GNOME 43, offers a polished and modern interface that is both intuitive and visually appealing. Moreover, Debian 12 includes a variety of accessibility tools and features, making it a suitable choice for users with disabilities.

Performance enhancements are also a significant part of Debian 12. The system has been optimized to run efficiently on both modern and older hardware, making it an excellent choice for extending the life of legacy systems. The inclusion of the Linux kernel 6.1 brings improved hardware support, better power management, and enhanced performance across various workloads. This makes Debian 12 a versatile operating system that can adapt to different environments, from low-power devices to high-performance servers.

1. Processor (CPU):
- A 64-bit (amd64) processor is recommended.
- Older 32-bit (i386) processors are also supported, but with reduced performance and capability compared to 64-bit processors.

2. Memory (RAM):
- Minimum: 512 MB of RAM.

- Recommended: 2 GB of RAM or more for a more comfortable experience, especially when running modern desktop environments and applications.

3. Storage:
- Minimum: 10 GB of available disk space.
- Recommended: 20 GB or more for additional software and personal files, especially if you plan to install a desktop environment and use the system for regular tasks.

4. Graphics:
- Basic VGA graphics adapter for a command-line installation.
- For a graphical installation and desktop environment, a graphics card that supports a resolution of at least 800x600 pixels.

5. Network:
- Network interface card (NIC) for internet access, which is essential for downloading updates and additional software packages.

6. Input Devices:
- Keyboard and mouse or other pointing devices.

For a more comfortable and versatile experience, especially when using a graphical desktop environment like GNOME or KDE Plasma, it is recommended to have a system with at least a dual-core processor, 4 GB of RAM, and 30 GB of available disk space. This ensures that the system can handle multitasking and run modern applications smoothly. Additionally, having a network connection during installation allows the installer to download the latest updates and software packages, further enhancing the system's functionality and security

4.2 Programming Language and Software:-

- **Spyder Information**

Spyder is an open-source IDE tailored for scientific and engineering development in Python. It provides a range of features designed to streamline data analysis and visualization, including:

- **Editor**: A powerful code editor with syntax highlighting, code completion, and error checking.
- **IPython Console**: An interactive console that supports inline plotting and rich media output.
- **Variable Explorer**: A tool to inspect and interact with variables in your workspace.
- **Documentation Viewer**: Integrated help and documentation for Python libraries.
- **Debugging Tools**: Features for setting breakpoints and stepping through code.

System Requirements for Linux (Debian)

Minimum Requirements:

- **Processor**: Any modern processor (Intel, AMD)
- **RAM**: 4 GB
- **Disk Space**: 100 MB for the application; additional space required for dependencies and project files
- **Python Version**: Python 3.6 or higher

Recommended Requirements:

- **Processor**: Multi-core processor
- **RAM**: 8 GB or more
- **Disk Space**: 200 MB or more, depending on the number of additional packages and project files
- **Python Version**: Python 3.8 or higher (recommended for better compatibility with newer packages)
- **Additional Packages**: Ensure that you have dependencies such as `numpy`, `pandas`, `matplotlib`, and `scipy` installed.

- **Python:-**

Python is a high-level, interpreted programming language known for its simplicity and readability. Python emphasizes code readability and allows developers to express concepts in fewer lines of code than other programming languages.

Key Features:
1. Simple Syntax: Python's syntax is designed to be easy to read and write, making it accessible for beginners and allowing developers to focus on solving problems rather than dealing with complex language syntax.

2. Interpreted Language: Python code is executed line by line, which simplifies debugging and testing. This interpreted nature allows for quick development cycles and immediate feedback.

3. Versatile and General-Purpose: Python is used in a wide range of applications, including web development, data analysis, artificial intelligence, scientific computing, and automation. Its versatility makes it suitable for both small scripts and large-scale applications.

4. Extensive Standard Library: Python comes with a comprehensive standard library that supports many common programming tasks, such as file I/O, system calls, and networking. This extensive library reduces the need for external dependencies.

5. Cross-Platform Compatibility: Python runs on various operating systems, including Windows, macOS, and Linux. This cross-platform support ensures that Python code can be developed and executed across different environments with minimal modifications.

6. Object-Oriented and Functional Programming: Python supports both object-oriented programming (OOP) and functional programming paradigms. This flexibility allows developers to choose the most appropriate approach for their specific tasks.

7. Large Community and Ecosystem: Python has a vibrant community that contributes to a rich ecosystem of third-party libraries and frameworks. Popular libraries such as NumPy, pandas, TensorFlow, and Django extend Python's capabilities and simplify complex tasks.

8. Ease of Learning: Python's straightforward syntax and clear structure make it an excellent language for beginners. Its readability and the availability of extensive educational resources contribute to a gentle learning curve.

9. Integration Capabilities: Python can easily integrate with other languages and technologies. It can be used to script and automate tasks in conjunction with other software, and it supports integration with databases, web services, and other tools.

Python's blend of simplicity, power, and versatility has made it one of the most popular programming languages in the world, widely used in both academic research and commercial applications.

- **Arduino IDE**

The Arduino Integrated Development Environment (IDE) is a software platform used for programming Arduino boards.

Features:
Code Editor: Provides a simple text editor where you can write and edit your Arduino sketches (programs). It includes features like syntax highlighting, automatic indentation, and basic code completion.

Compiler: Translates your code into machine language that the Arduino can understand. It also checks for errors and warns you of any issues in your code.

Uploader: Sends the compiled code to the Arduino board via a serial connection. This process is usually done through a USB cable.

Serial Monitor: Allows you to view and interact with data being sent from the Arduino board to your computer. This is useful for debugging and for receiving real-time data from your sensors.

Additional Features:
Library Manager: Lets you include and manage libraries that provide pre-written code to help you with various tasks like controlling motors or reading sensor data.

Board Manager: Enables you to select and configure different Arduino boards and compatible hardware.

Example Sketches: Provides a variety of pre-written example programs to help you get started with different Arduino features and components.

User Interface:
The IDE's interface is designed to be user-friendly, especially for beginners. It features a clean layout with a menu bar, a toolbar with commonly used functions, and panels for code editing and serial communication.

Arduino IDE is designed to simplify the process of writing, uploading, and debugging code on Arduino boards, making it accessible for both beginners and experienced users.

- **Software Packages Used:-**

Python packages :-

PySerial is a Python library that encapsulates access for the serial port. It provides backend support for serial communication, allowing Python scripts to read from and write to serial devices such as Arduino, serial consoles, and more.

Key Features of PySerial:

1. **Cross-Platform Support**: Works on Windows, Linux, and macOS.
2. **Easy to Use**: Simplifies reading and writing data to serial ports.
3. **Timeout Handling**: Allows setting timeouts for reading operations.
4. **Binary Data Handling**: Supports reading and writing binary data.
5. **Line-Oriented Protocols**: Provides helper functions to read and write lines of text.

Installation Procedure for PySerial via `apt` on Debian:

1. Update the Package List:

Open a terminal and run the following command to update the package list:

```
sudo apt update
```

2. Install PySerial:

Use `apt` to install PySerial directly:

```
sudo apt install python3-serial
```

3. Verify Installation:

You can verify that PySerial is installed correctly by opening a Python shell and importing the library:

```
python3 -c "import serial; print(serial.__version__)"
```

This command should print the installed version of PySerial.

Troubleshooting:

Permission Denied: If you encounter a permission denied error when accessing the serial port, you may need to add your user to the `dialout` group:

```
sudo usermod -aG dialout $USER
```

Log out and log back in for the changes to take effect.

Serial Port Identification: To identify the correct serial port, you can use the `dmesg | grep tty` command after plugging in your serial device.

Arduino Libraries:-

#include <Arduino.h> is a directive in the Arduino programming environment that includes the core Arduino functions and libraries. This file is essential for writing Arduino sketches, as it provides the basic functionality required for interacting with the microcontroller hardware, such as digital and analog input/output, timing functions, and serial communication.

Key Points About #include <Arduino.h>:

Core Functionality: It includes definitions for core functions such as setup(), loop(), digitalRead(), digitalWrite(), analogRead(), analogWrite(), delay(), millis(), and many more.

Compatibility: Ensures compatibility across different Arduino boards by abstracting hardware-specific details. This allows sketches to be portable across different Arduino models.

Initialization: Handles the necessary initialization required for the microcontroller, such as setting up timers, configuring input/output pins, and initializing serial communication.

Standard Library: It includes a set of standard libraries and utilities that are commonly used in Arduino programming, such as Serial, String, Wire, and EEPROM.

CHAPTER 5
PROPOSED SYSTEM

Purpose:

This project report delves into the development and implementation of an advanced antenna pointing and control system designed specifically for enhancing RFI monitoring capabilities.

The core objective of this project is to design and develop a sophisticated system that enables precise directional control of an antenna. This is crucial for effectively locating and analyzing sources of RFI, which can otherwise obscure valuable data and disrupt critical communications.

Our approach integrates various electronic components and programming techniques to achieve seamless control over the antenna's orientation. The system utilizes an Arduino microcontroller as the central hub for managing the control mechanisms. This microcontroller communicates via serial communication to coordinate the operation of an L298N motor driver, which in turn controls the movement of the antenna through a series of gears.

To ensure accurate positioning, the system incorporates a 2500 PPR (pulses per revolution) optical encoder, which provides high-resolution feedback on the antenna's position. This feedback is crucial for precise adjustments and maintaining the antenna in the desired direction. The entire system is orchestrated through a custom-designed graphical user interface (GUI), which allows users to easily input and adjust directional parameters for the antenna.

The GUI, developed using Python programming, offers an intuitive interface for controlling the antenna's pointing direction. It integrates with the Arduino through serial communication, enabling real-time adjustments and monitoring of the antenna's position. The combination of these technologies ensures a robust and user-friendly system capable of meeting the demands of modern RFI monitoring.

This project represents a significant advancement in antenna control systems, leveraging a blend of electronics, programming, and mechanical components to provide precise and efficient RFI monitoring capabilities.

CHAPTER 6
DESIGN AND WORKFLOW DIAGRAM

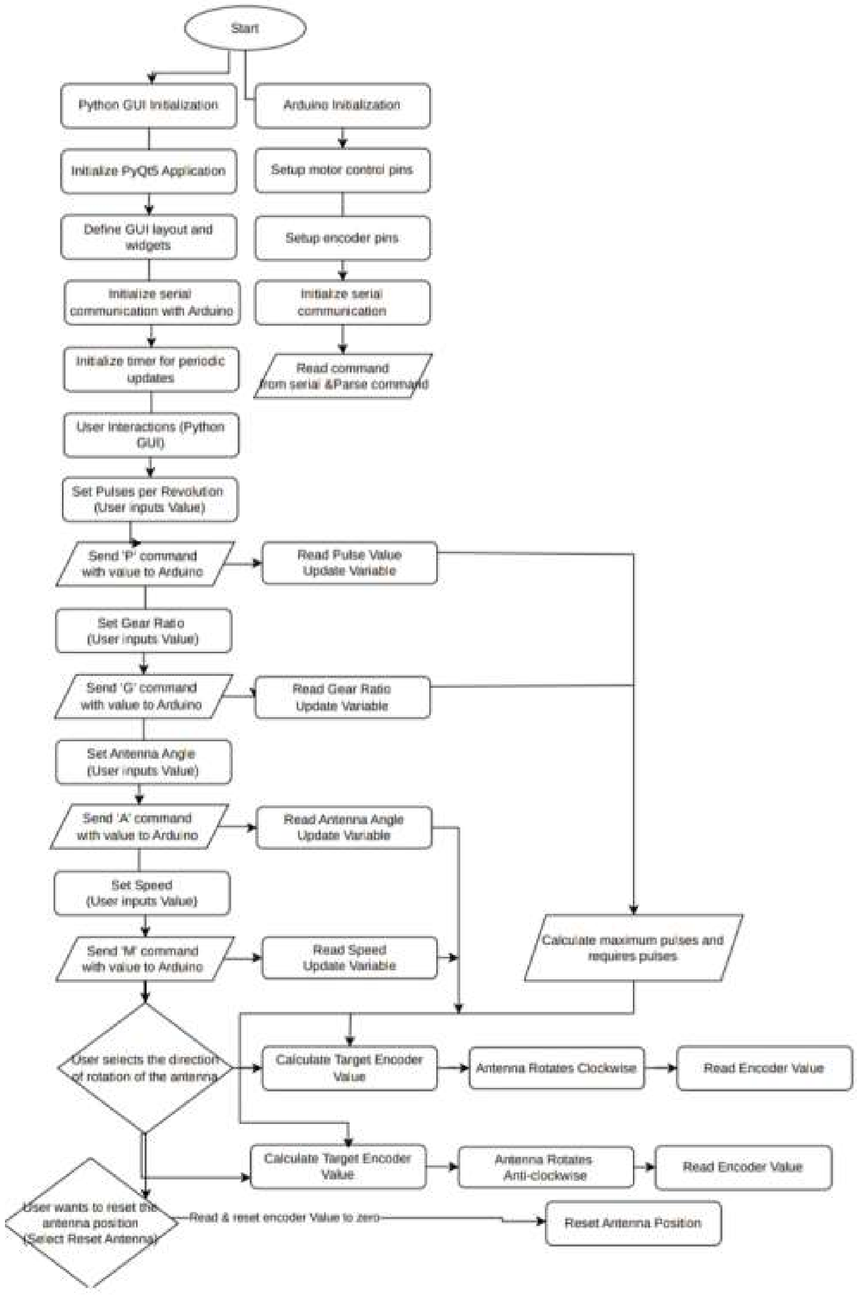

Start
Python GUI Initialization
Arduino Initialization
Initialize PyQt5 Application
Setup motor control pins
Define GUI layout and widgets
Setup encoder pins
Initialize serial communication with Arduino
Initialize serial communication
Initialize timer for periodic updates
Read command from serial &Parse command
User Interactions (Python GUI)
Set Pulses per Revolution (User inputs Value)
Send 'P' command with value to Arduino
Read Pulse Value Update Variable
Set Gear Ratio (User inputs Value)
Send 'G' command with value to Arduino
Read Gear Ratio Update Variable
Set Antenna Angle (User inputs Value)
Send 'A' command with value to Arduino
Read Antenna Angle Update Variable
Set Speed (User inputs Value)
Send 'M' command with value to Arduino
Read Speed Update Variable
Calculate maximum pulses and requires pulses
User selects the direction of rotation of the antenna
Calculate Target Encoder Value
Antenna Rotates Clockwise
Read Encoder Value
Calculate Target Encoder Value
Antenna Rotates Anti-clockwise
Read Encoder Value
User wants to reset the antenna position (Select Reset Antenna)
Read & reset encoder Value to zero
Reset Antenna Position

Fig. Workflow Diagram for the Proposed System

This diagram represents the workflow of the RFI Antenna Control System application. The process begins with initialization, where Python and Arduino components are set up, establishing the user interface and connection via PySerial. Following initialization, the Main Control Window is displayed to the user. The system then enters a continuous loop, executing tasks such as rotating the antenna to a specified position. During this phase, user interactions are managed, allowing the user to set parameters like speed and angle as needed. The process concludes with the termination of the application.

CHAPTER 7
LOGIC OF THE PROGRAM

Logic of the python script:-

1. Initialization (`__init__`):

Purpose: Initializes the main GUI window, sets up the serial communication, and starts a timer for periodic tasks.

- Steps:
  1. Calls `super().__init__()` to initialize the `QMainWindow`.
  2. Assigns the `motor_control` object (currently `None` in this script) to an instance variable.
  3. Initializes the PWM value as `None`.
  4. Calls `initUI()` to build the GUI layout and elements.
  5. Calls `init_serial()` to set up the serial communication.
  6. Calls `init_timer()` to start the timer that periodically checks for encoder values.

2. Serial Communication (`init_serial`):

- Purpose: Sets up the serial communication to interact with the Arduino.
- Steps:
  1. Opens the serial port (`/dev/ttyUSB1`) at a baud rate of 9600, with a timeout of 1 second.

3. Timer Initialization (`init_timer`):

- Purpose: Initializes and starts a timer to periodically update the encoder value.
- Steps:
  1. Creates a `QTimer` object.
  2. Connects the timer's `timeout` signal to the `update_encoder_value()` method.
  3. Starts the timer to trigger every 1000 milliseconds (1 second).

4. Send Serial Command (`send_serial_command`):

- Purpose: Sends commands to the Arduino through the serial port.
- Steps:
  1. Checks if the serial port is open.
  2. Sends the command to the Arduino.
  3. Logs the action of sending the command.

5. Update Encoder Value (`update_encoder_value`):

- Purpose: Reads data from the serial port and updates the encoder value on the GUI.
- Steps:
  1. Checks if there is data waiting on the serial port.
  2. Reads the data from the serial port, decodes it, and strips any extra whitespace.
  3. Updates the `encoder_value_label` with the new value if data is received.

6. GUI Setup (`initUI`):

- Purpose: Sets up the main GUI layout and widgets.
- Steps:
  1. Sets the window title, size, and dark mode palette.
  2. Creates a `QWidget` as the central widget for the main window.
  3. Creates three rows of widgets:
     -First Row: Inputs for setting pulses per revolution, motor-to-shaft ratio, shaft-to-antenna ratio, and antenna angle.
     - Second Row: Direction controls, speed control, and encoder value display.
     - Third Row: Buttons to submit parameters, reset all parameters, and reset the antenna position.

4. Adds a spacer to the bottom of the main layout for better visual alignment.
5. Calls `read_motor_settings()` to initialize input fields with saved values from a file.

7. Reset Antenna Position (`reset_antenna_position`):
- Purpose: Resets the antenna position to an encoder count of 0.
- Steps:
  1. Clears the angle input field.
  2. Sends a reset command (`R`) to the Arduino.
  3. Displays a confirmation message and logs the action.

8. Read Motor Settings (`read_motor_settings`):
- Purpose: Reads saved motor settings from a file and updates the GUI with these settings.
- Steps:
  1. Checks if the settings file exists.
  2. Reads the file and updates the input fields with the saved settings.

9. Submit Pulses per Revolution (`submit_pulses`):
- Purpose: Submits the pulses per revolution to the Arduino.
- Steps:
  1. Reads the input from the `pulses_entry` field.
  2. Sends the pulses as a command (`P:<value>`) to the Arduino.
  3. Logs the action.

10. Submit Gear Ratio (`submit_gear_ratio`):
- Purpose: Submits the motor-to-shaft gear ratio to the Arduino.
- Steps:
  1. Reads the input from the `gear_ratio_entry` field.
  2. Sends the gear ratio as a command (`G:<value>`) to the Arduino.
  3. Logs the action.

11. Submit Encoder Ratio (`submit_shaft_ratio`):
- Purpose: Submits the shaft-to-antenna ratio to the Arduino.
- Steps:
  1. Reads the input from the `shaft_ratio_entry` field.
  2. Sends the shaft ratio as a command (`E:<value>`) to the Arduino.
  3. Logs the action.

12. Submit Antenna Angle (`submit_angle`):
- Purpose: Submits the desired antenna angle to the Arduino.
- Steps:
  1. Reads the input from the `pulse_entry` field.
  2. Sends the angle as a command (`A:<value>`) to the Arduino.
  3. Logs the action.

13. Set Motor Speed (`set_speed`):
- Purpose: Sets the speed of the motor by sending a PWM value to the Arduino.
- Steps:
  1. Reads the input from the `speed_entry` field.
  2. Validates that the speed is within the acceptable range (0-255).
  3. Sends the speed as a command (`M:<value>`) to the Arduino.
  4. Updates the speed label and logs the action.

14. Forward Movement (`forward`):
- Purpose: Sends a command to move the motor forward (clockwise).
- Steps:
  1. Sends the forward command (`F`) to the Arduino.
  2. Displays a confirmation message and logs the action.

15. Backward Movement (`backward`):
- Purpose: Sends a command to move the motor backward (anti-clockwise).
- Steps:
  1. Sends the backward command (`B`) to the Arduino.
  2. Displays a confirmation message and logs the action.

16. Read Encoder Value (`read_encoder`):
- Purpose: Sends a command to read the current encoder value from the Arduino.
- Steps:
  1. Sends the encoder read command (`E`) to the Arduino.
  2. Logs the action.

17. Direction Count (`direction_count`):
- Purpose: Sends a command to get the current direction count from the Arduino.
- Steps:
  1. Sends the direction count command (`D`) to the Arduino.
  2. Displays a confirmation message and logs the action.

18. Reset All Parameters (`reset_all`):
- Purpose: Resets all input fields in the GUI to their default (empty) state.
- Steps:
  1. Clears all input fields.
  2. Displays a confirmation message and logs the action.

19. Submit All Parameters (`submit_parameters`):
- Purpose: Submits all input parameters to the Arduino.
- Steps:
  1. Calls the methods to submit pulses, gear ratio, shaft ratio, antenna angle, and speed.
  2. Logs the action.

20. Dark Mode Settings (`setDarkMode`):
- Purpose: Sets the GUI to use a dark color palette.
- Steps:
  1. Configures the `QPalette` with dark colors for various GUI elements.
  2. Applies the palette to the main window.

21. Logging (`log_action`):
- Purpose: Logs user actions and system events to a text file.
- Steps:
  1. Opens the log file in append mode.
  2. Writes a timestamped log entry with the provided action description.

Logic of the Arduino Script:-
This Arduino code is designed to control a motor for rotating an antenna with precise angle measurements using an encoder. The code communicates with a Python application to receive commands and data, controlling the motor's speed, direction, and position based on the inputs provided. Here’s a step-by-step explanation of the logic:

1. Initial Setup
- Motor Control Pins: The motor is controlled using three pins:
  - `enablePin` (PWM): Controls motor speed.
  - `int1Pin` and `int2Pin`: Control the direction of the motor rotation.

- Encoder Pins: The encoder, which tracks the antenna's rotation, is connected to `encoderPin1` and `encoderPin2`.

- Initialization:
  - Begin serial communication at 9600 baud rate.
  - Set the motor control pins (`enablePin`, `int1Pin`, `int2Pin`) as outputs.
  - Set encoder pins (`encoderPin1`, `encoderPin2`) as inputs with internal pull-up resistors enabled.
  - Attach interrupts to the encoder pins to detect changes in rotation and update the encoder count accordingly.

2. Pulse Conversion (Function: `conversion`)
- Converts pulses received from Python into the maximum encoder value for a full 360-degree rotation.
- Considers the gear ratio and shaft ratio in the calculation.
- Calculates the number of pulses required to rotate the antenna by a specified angle.

3. Main Loop (`loop` function)
- Continuously monitors serial input for commands from the Python application.
- Command Handling:
  - 'A’: Set the antenna angle.
  - 'P': Set the pulse count per revolution.
  - 'G': Set the motor-to-shaft gear ratio.
  - 'E': Set the shaft-to-antenna gear ratio.
  - 'F': Move the antenna forward (clockwise direction).
  - 'B': Move the antenna backward (anti-clockwise direction).
  - 'D': Check the direction of the antenna’s rotation.
  - 'R': Reset the antenna position to zero.
  - 'S': Stop the motor.
  - 'M': Set the motor speed.

4. Encoder Value Update (Function: `updateEncoder`)
- Tracks changes in the encoder’s output using interrupts.
- Determines the direction of rotation and increments or decrements the `encoderValue` based on the encoder signals.

5. Motor Control Functions
- Forward (`forward`):
  - Rotates the antenna clockwise until the `encoderValue` reaches the target value corresponding to the requested angle.
- Backward (`backward`):
  - Rotates the antenna anti-clockwise until the `encoderValue` reaches the target value.
- Stop Motor (`stopMotor`):

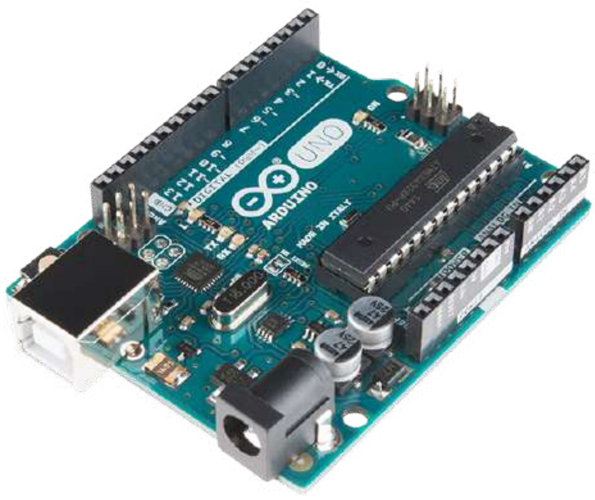
  - Stops the motor by setting both direction pins low.
- Direction Check (`directionCheck`):
  - Determines and prints the direction of rotation by comparing the current encoder value with the previous one.
- Reset (`reset`):
  - Resets the antenna position to zero by rotating it back to the original position.

6. Communication
- The code communicates with the Python application via serial communication to receive commands and data, allowing the user to control the antenna's movement and check its status.

This setup enables precise control of the antenna's position through a motor, ensuring it can rotate to specific angles as required by the user. The gear ratios and encoder values are considered for accurate rotation, and the direction and position can be monitored and adjusted in real-time.

Chapter 8
COMPONENTS USED

1. Arduino UNO R3 (ATMEGA328P):-



Arduino UNO is a microcontroller board based on the ATmega328P. It has 14 digital input/output pins (of which 6 can be used as PWM outputs), 6 analog inputs, a 16 MHz ceramic resonator, a USB connection, a power jack, an ICSP header and a reset button. It contains everything needed to support the microcontroller.

Memory

- AVR CPU at up to 16 MHz
- 32 kB Flash
- 2 kB SRAM
- 1 kB EEPROM

Peripherals

- 2x 8-bit Timer/Counter with a dedicated period register and compare channels
- 1x 16-bit Timer/Counter with a dedicated period register, input capture and compare channels
- 1x USART with fractional baud rate generator and start-of-frame detection
- 1x controller/peripheral Serial Peripheral Interface (SPI)
- 1x Dual mode controller/peripheral I2C
- 1x Analog Comparator (AC) with a scalable reference input
- Six PWM channels
- Interrupt and wake-up on pin change

ATMega16U2 Processor

- 8-bit AVR® RISC-based microcontroller

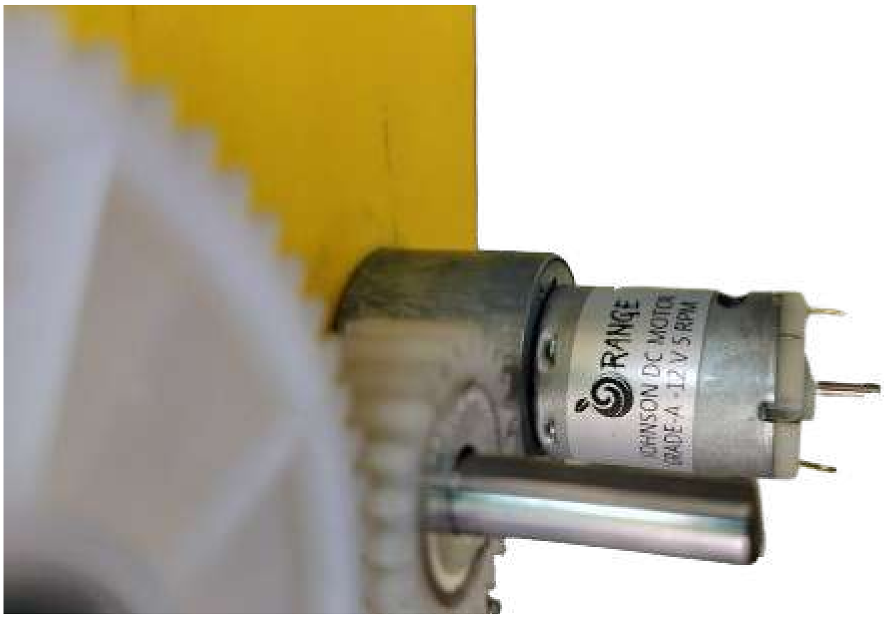
Memory

- 16 kB ISP Flash
- 512B EEPROM
- 512B SRAM
- debugWIRE interface for on-chip debugging and programming

Power

- 2.7-5.5 volts

2. **12 Volt Johnson Geared DC Motor:-**



10RPM 12V DC Johnson high torque geared motors for robotics applications. It gives a massive torque of 120Kgcm. The motor comes with metal gearbox and off-centered shaft.

Features:

- 10RPM 12V DC motors with Metal Gearbox and Metal Gears
- 18000 RPM base motor
- 6mm Dia shaft with M3 thread hole
- Gearbox diameter 37 mm.
- Motor Diameter 28.5 mm
- Length 63 mm without shaft
- Shaft length 30mm
- 180gm weight
- 120kgcm Holding Torque
- No-load current = 800 mA, Load current = upto 7.5 A(Max)

3. Pro-Range 2500 PPR ABZ 3-Phase Incremental Optical Rotary Encoder

A rotary encoder is a type of position sensor which is used for determining the angular position of a rotating shaft. It generates an electrical signal, either analog or digital, according to the rotational movement.

Orange 2500 PPR Incremental Optical Rotary Encoder is a hi-resolution optical encoder with quadrature outputs for increment counting. It will give 10000 transitions per rotation between outputs A and B. whereas the Z phase will produce one transition per rotation. A quadrature decoder is required to convert the pulses to an up count. The Encoder is built to Industrial grade.

The Encoder comes with a Standard 2-meter long cable which can be extended with extra cable if needed.

Phase A	White	Quadrature encoded output A
Phase B	Green	Quadrature encoded output B
Phase Z	Yellow	Quadrature encoded output Z
VCC	Red	VCC should be connected to +ve 5V of supply
GND	Black	The ground should be connected to negative the supply
Shield	Golden	The shield should be connected to GND

Features :
- High cost-efficient advantages.
- Incremental rotary encoder internal adopts ASIC devices
- High reliability, long life
- Anti-jamming performance
- Small size, lightweight, compact structure
- Easy installation
- Stainless steel shaft, High resolution, High quality, line interface with waterproof protection

4. **12 volt 5 amp power supply:-**

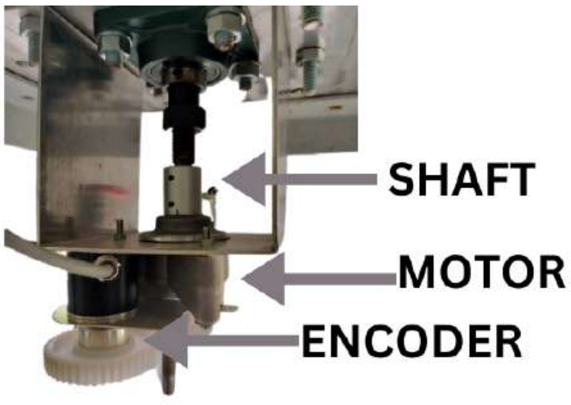
Power your devices efficiently with our 12 Volt 5 Amp Adapter, designed to deliver reliable performance for a wide range of applications. This adapter provides a stable 12V output at 5 Amps, ensuring consistent power for your electronics.

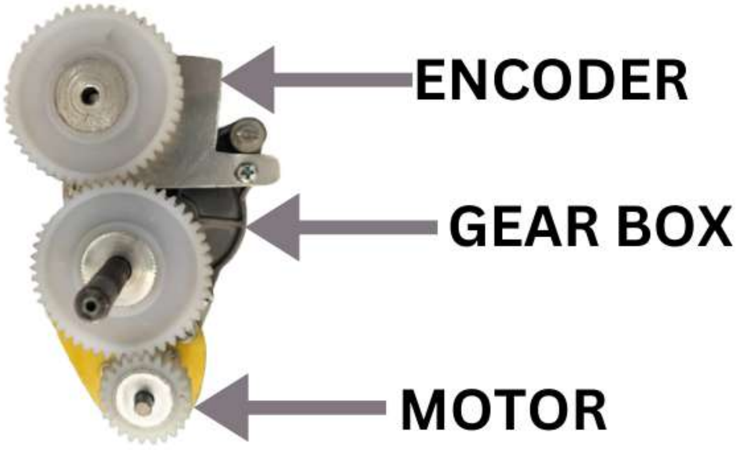
Key Features:

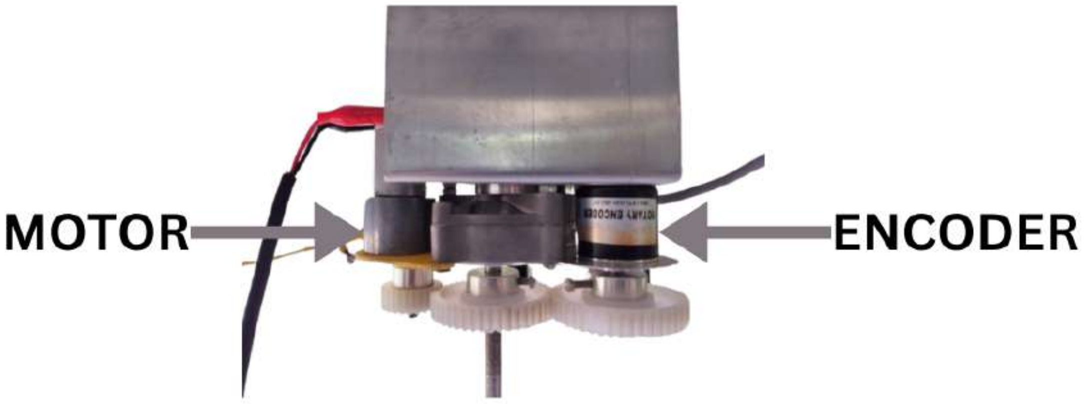
- **Model Name:** 12 Volt 5 Amp
- **Connector Pin Type:** 2.5mm x 5.5mm, perfect for compatibility with various devices.
- **Power Input:** 220V, suitable for use with standard household outlets.
- **Output Current:** 5 Amp, offering ample current for power-hungry devices.
- **Cable Length:** 5 meters, giving you flexibility in placement and use.
- **Short Circuit Protection:** Built-in safety feature to protect your devices from potential damage due to short circuits.

Model Id	• 12 Volt 5 Amp Adapter/ Power Charger
Model Name	• 12 Volt 5Amp
Connector Pin Type	• 2.5mm x 5.5mm
Power Input	• 220V
Output Current (A)	• 5Amp A
Cable Length (m)	• 5 m
Short Circuit Protection	• Yes

5. **Gear Box:-**







Fig, Gear box for Antenna rotation

The above shown gear box has a 12 Volt Johnson DC Motor, 2500 PPM Optical Incremental Encoder, Gears, and a shaft.

Specification:-

Motor to Shaft ratio = 5.8

Encoder to Shaft Ratio = 1.12

Calculating the Angular Speed in the Degree per Seconds:-

To calculate the angular speed in revolutions per minute (RPM) using PWM (Pulse Width Modulation) and the time it takes to rotate 90 degrees, you can follow these steps:

Step 1: Calculate the Angular Speed in Degrees per Second

1. **Given:**

- Time for 90 degrees=t90 seconds

2. **Calculate Degrees per Second:**
    - Angular Speed in Degrees per Second=90/t90

Step 2: Convert Degrees per Second to Revolutions per Minute (RPM)

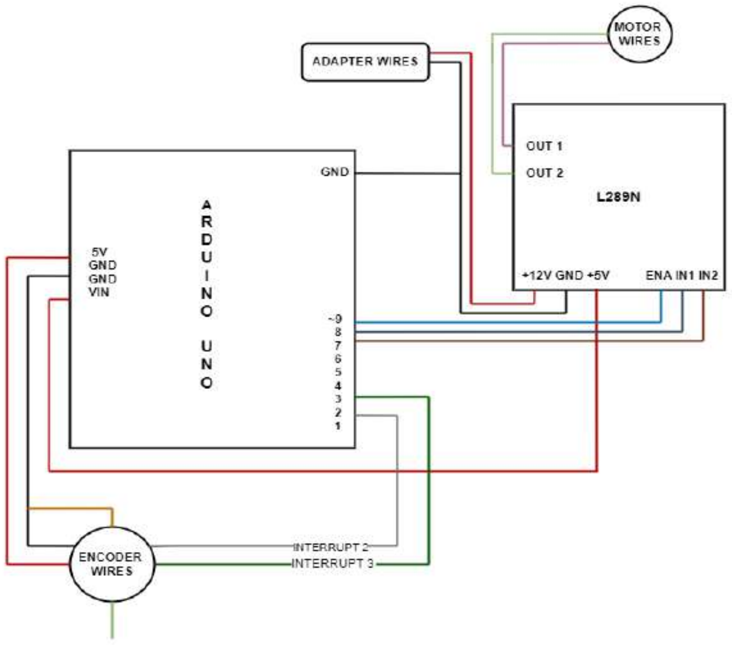
1. **Convert Degrees per Second to Revolutions per Second:**
    - Revolutions per Second=Degrees per Second / 360
    - Revolution per Seconds = 90 / t90*360
2. **Convert Revolutions per Second to RPM:**
    - RPM=Revolutions per Second × 60
    - RPM=90*60/t90*360 = 90/t90*60
    - RPM=15 / t90

Final Formula:

RPM=15/t90

Where t90 is the time in seconds to complete a 90-degree rotation.

CHAPTER 9
CIRCUIT DIAGRAM AND CONNECTIONS



Circuit Description

Components and Connections:

1. Arduino Uno:
- Digital Pins:
  - Pin 3: Connected to the white wire (Pulse A) of the encoder.
  - Pin 4: Connected to the green wire (Pulse B) of the encoder.
  - Pin 8: Connected to IN1 of the L298N motor driver.
  - Pin 7: Connected to IN2 of the L298N motor driver.
  - Pin 9: Connected to the ENA (Enable) pin of the L298N motor driver.

- Power Pins:
  - 5V Pin: Connected to the +5V input of the L298N motor driver.
  - GND Pin: Connected to the ground of the power supply adapter.

2. Encoder:
- Pulse A (White Wire): Connected to Arduino Pin 3.
- Pulse B (Green Wire): Connected to Arduino Pin 4.

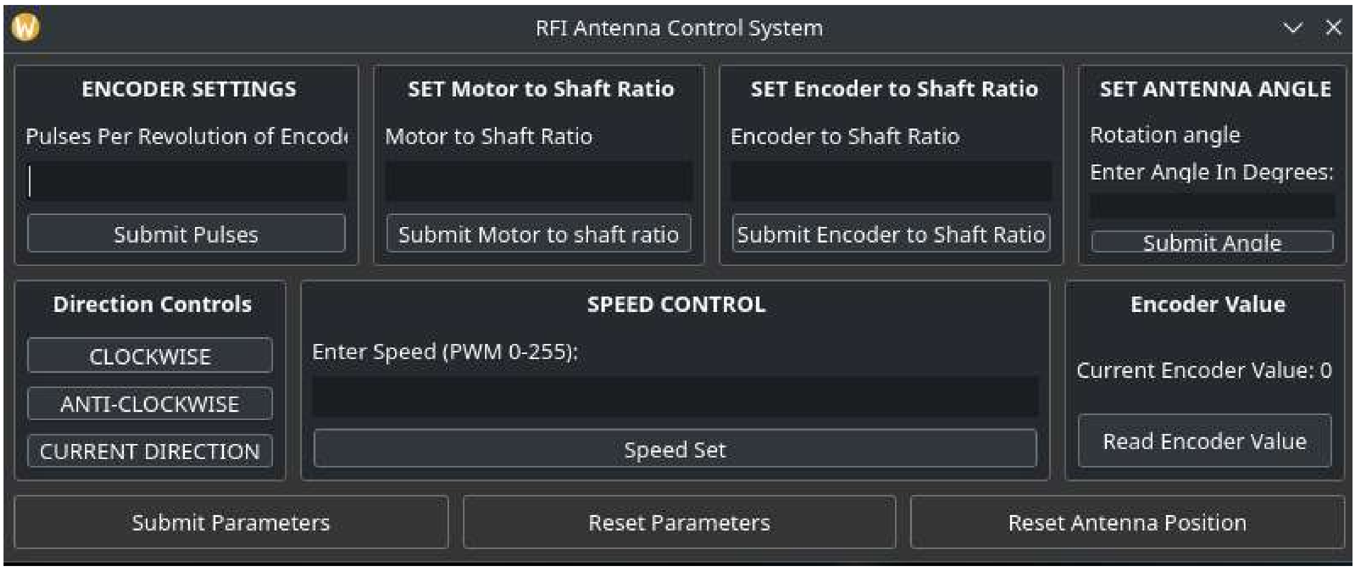
- +5V (Red Wire): Connected to the +5V pin on the Arduino.
  - Ground (Black Wire): Connected to the GND pin on the Arduino.

3. L298N Motor Driver:
- Motor Output Pins:
  - Motor 1: Connected to the motor terminals.
  - Motor 2: Connected to the motor terminals.
- Enable Pin (ENA): Connected to Arduino Pin 9 for PWM control.
- IN1: Connected to Arduino Pin 8.
- IN2: Connected to Arduino Pin 7.
- +5V Input: Connected to the +5V pin on the Arduino.

4. Power Supply:
- 12V Adapter:
  - +12V Output: Connected to the motor power input of the L298N motor driver.
  - GND Output: Connected to the GND pin on the Arduino to ensure a common ground.

Operation:
- The encoder’s Pulse A and Pulse B signals are used by the Arduino to monitor the motor’s rotation and speed.
- The Arduino controls the L298N motor driver through digital pins to manage the motor's direction and speed.
- PWM signal from Arduino Pin 9 is used to control the motor’s speed via the L298N motor driver’s enable pin.
- The 12V power supply provides necessary voltage to the motor driver, while the Arduino is powered through its own USB connection or an external 5V source, which is also connected to the L298N.

CHAPTER 10
RFI ANTENNA CONTROL SYSTEM APPLICATION



Encoder Configuration Panel: This section allows precise configuration of the encoder parameters. The encoder count is specified here, set at 2500 pulses per revolution (PPR). The motor-to-shaft ratio is configured to 5.8, enabling the system to translate motor revolutions into corresponding shaft rotations. Additionally, the encoder-to-shaft ratio is set at 1.12 to ensure accurate measurement and control of angular displacement.

Antenna Angle Setting: This field requires the user to input the desired angular displacement for the antenna. This input directly dictates the target position for the antenna, enabling precise pointing based on the calculated ratios.

Direction Control Panel: This panel facilitates the directional control of the motor's rotation. The user can select either clockwise or counterclockwise rotation, depending on the operational requirements. The "Current Direction" button displays the current motor rotation direction in the Serial Monitor panel, providing real-time feedback for monitoring.

Speed Control Panel: The PWM (Pulse Width Modulation) value is entered here, with a valid range between 60 and 255. This range has been determined based on prior motor performance observations, ensuring optimal motor speed and response.

Encoder Value Display: This panel continuously monitors and displays the current encoder value, reflecting the real-time rotational position of the motor shaft.

Parameter Submission: The "Submit Parameter" button consolidates all configured values and transmits them to the Arduino via serial communication from the Python script. This ensures synchronized execution of the motor control algorithm.

Reset Functions:

- The "Reset Parameter" button clears all input fields within the GUI, allowing for fresh configuration inputs.
- The "Reset Antenna Position" button commands the system to return the antenna to its default parking position, resetting the antenna's orientation to zero degrees.

CHAPTER 11
PROGRAM CODE

10.1 Python Program:-

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Thu Aug  8 14:36:22 2024

@author: madhav & shreya
"""

#importing all the library files needed
import sys
import os
from PyQt5.QtWidgets import (
    QApplication, QMainWindow, QWidget, QVBoxLayout, QHBoxLayout, QLabel,
    QLineEdit, QPushButton, QGroupBox, QSpacerItem, QSizePolicy, QMessageBox
)
from PyQt5.QtCore import Qt, QTimer
from PyQt5.QtGui import QPalette, QColor
import serial
import datetime

# Define the file path for saving motor settings and logs
file_path =
'/home/madhav/Pictures/GMRT_RFI_PROJECT_GUI_CHANGES_DONE/motor_settings.txt'
log_file_path = '/home/madhav/Pictures/GMRT_RFI_PROJECT_GUI_CHANGES_DONE/log.txt'

# GUI main window
class MotorControlGUI(QMainWindow):
    def __init__(self, motor_control):
        super().__init__()
        self.motor_control = motor_control
        self.pwm_value = None  # Initialize PWM value as None
        self.initUI()
        self.init_serial()
        self.init_timer()

    # For serial communication, declare ports and baudrate here.
    def init_serial(self):
        self.serial_port = serial.Serial('/dev/ttyUSB0', 9600, timeout=1)  # Adjust COM port as needed

    # For periodically updating the encoder value
    def init_timer(self):
        self.timer = QTimer()
        self.timer.timeout.connect(self.update_encoder_value)
        self.timer.start(1000)  # Check every 1000 ms (1 second)

    # For sending commands to Arduino
```

```
def send_serial_command(self, command):
    if self.serial_port.is_open:
        self.serial_port.write(command.encode())
        self.log_action(f'Sent command: {command}')
    else:
        QMessageBox.critical(self, "Serial Port Error", "Serial port is not open.")

# Update encoder value from serial port
def update_encoder_value(self):
    if self.serial_port.in_waiting:
        data = self.serial_port.readline().decode().strip()
        if data:
            self.encoder_value_label.setText(f'Current Encoder Value: {data}')


# Creating the GUI of the Antenna Controlling
def initUI(self):
    self.setWindowTitle('RFI Antenna Control System')
    self.setGeometry(100, 100, 800, 600)
    self.setFixedSize(800, 300)
    self.setDarkMode()

    central_widget = QWidget()
    self.setCentralWidget(central_widget)
    main_layout = QVBoxLayout()
    central_widget.setLayout(main_layout)

    # First Row: Set Pulses per Revolution, Set Gear Ratio, Set Antenna Angle
    first_row_layout = QHBoxLayout()
    main_layout.addLayout(first_row_layout)

    # Set Pulses per Revolution
    pulses_group = QGroupBox('ENCODER SETTINGS')
    pulses_group.setStyleSheet("""
                    QGroupBox {
                        font-weight: bold;
                        font-size: 13px;
                        }
                    """)
    pulses_layout = QVBoxLayout()
    pulses_group.setLayout(pulses_layout)

    pulses_title_label = QLabel('Pulses Per Revolution of Encoder')
    pulses_layout.addWidget(pulses_title_label)

    self.pulses_entry = QLineEdit()
    pulses_layout.addWidget(self.pulses_entry)

    self.submit_pulses_button = QPushButton('Submit Pulses')
    self.submit_pulses_button.clicked.connect(self.submit_pulses)
    pulses_layout.addWidget(self.submit_pulses_button)
```

```
first_row_layout.addWidget(pulses_group)

# Set Motor to Shaft Ratio
motor_to_shaft_group = QGroupBox('SET Motor to Shaft Ratio')
motor_to_shaft_group.setStyleSheet("""
                        QGroupBox {
                            font-weight: bold;
                            font-size: 13px;
                            }
                        """)
motor_to_shaft_layout = QVBoxLayout()
motor_to_shaft_group.setLayout(motor_to_shaft_layout)

motor_to_shaft_title_label = QLabel('Motor to Shaft Ratio')
motor_to_shaft_layout.addWidget(motor_to_shaft_title_label)

self.gear_ratio_entry = QLineEdit()
motor_to_shaft_layout.addWidget(self.gear_ratio_entry)

self.submit_gear_ratio_button = QPushButton('Submit Motor to shaft ratio')
self.submit_gear_ratio_button.clicked.connect(self.submit_gear_ratio)
motor_to_shaft_layout.addWidget(self.submit_gear_ratio_button)

first_row_layout.addWidget(motor_to_shaft_group)

# Set Shaft to Antenna Ratio
shaft_to_antenna_group = QGroupBox('SET Encoder to Shaft Ratio')
shaft_to_antenna_group.setStyleSheet("""
                        QGroupBox {
                            font-weight: bold;
                            font-size: 13px;
                            }
                        """)
shaft_to_antenna_layout = QVBoxLayout()
shaft_to_antenna_group.setLayout(shaft_to_antenna_layout)

shaft_to_antenna_title_label = Qlabel('Encoder to Shaft Ratio')
shaft_to_antenna_layout.addWidget(shaft_to_antenna_title_label)

self.shaft_ratio_entry = QLineEdit()
shaft_to_antenna_layout.addWidget(self.shaft_ratio_entry)

self.submit_shaft_ratio_button = QPushButton('Submit Encoder to Shaft Ratio')
self.submit_shaft_ratio_button.clicked.connect(self.submit_shaft_ratio)
shaft_to_antenna_layout.addWidget(self.submit_shaft_ratio_button)

first_row_layout.addWidget(shaft_to_antenna_group)

# Set Antenna Angle
pulse_group = QGroupBox('SET ANTENNA ANGLE')
pulse_group.setStyleSheet("""
                        QGroupBox {
                                font-weight: bold;
                                font-size: 13px;
                                }
                            """)
pulse_layout = QVBoxLayout()
pulse_group.setLayout(pulse_layout)

pulse_title_label = QLabel('Rotation angle')
pulse_layout.addWidget(pulse_title_label)

pulse_label = QLabel('Enter Angle In Degrees:')
pulse_layout.addWidget(pulse_label)

self.pulse_entry = QLineEdit()
pulse_layout.addWidget(self.pulse_entry)

self.submit_angle_button = QPushButton('Submit Angle')
self.submit_angle_button.clicked.connect(self.submit_angle)
pulse_layout.addWidget(self.submit_angle_button)

first_row_layout.addWidget(pulse_group)

# Second Row: Direction Controls, Speed Controls, Encoder Value
second_row_layout = QHBoxLayout()
main_layout.addLayout(second_row_layout)

# Direction Controls
direction_group = QGroupBox('Direction Controls')
direction_group.setStyleSheet("""
                            QGroupBox {
                                font-weight: bold;
                                font-size: 13px;
                                }
                            """)
direction_layout = QVBoxLayout()
direction_group.setLayout(direction_layout)

self.forward_button = QPushButton('CLOCKWISE')
self.forward_button.clicked.connect(self.forward)
direction_layout.addWidget(self.forward_button)

self.backward_button = QPushButton('ANTI-CLOCKWISE')
self.backward_button.clicked.connect(self.backward)
direction_layout.addWidget(self.backward_button)

self.direction_button = QPushButton('CURRENT DIRECTION')
self.direction_button.clicked.connect(self.direction_count)
direction_layout.addWidget(self.direction_button)

second_row_layout.addWidget(direction_group)

# Speed Controls
```

```
motor_controls_group = QGroupBox('SPEED CONTROL')
motor_controls_group.setStyleSheet("""
                QGroupBox {
                    font-weight: bold;
                    font-size: 13px;
                    }
                """)
motor_controls_layout = QVBoxLayout()
motor_controls_group.setLayout(motor_controls_layout)

self.speed_label = QLabel('Enter Speed (PWM 0-255):')
motor_controls_layout.addWidget(self.speed_label)

self.speed_entry = QLineEdit()
motor_controls_layout.addWidget(self.speed_entry)

self.speed_set_button = QPushButton('Speed Set')
self.speed_set_button.clicked.connect(self.set_speed)
motor_controls_layout.addWidget(self.speed_set_button)

second_row_layout.addWidget(motor_controls_group)

# Encoder Value
encoder_group = QGroupBox('Encoder Value')
encoder_group.setStyleSheet("""
                    QGroupBox {
                        font-weight: bold;
                        font-size: 13px;
                        }
                    """)
encoder_layout = QVBoxLayout()
encoder_group.setLayout(encoder_layout)

self.encoder_value_label = QLabel('Current Encoder Value: N/A')
encoder_layout.addWidget(self.encoder_value_label)

self.read_encoder_button = QPushButton('Read Encoder Value')
self.read_encoder_button.clicked.connect(self.read_encoder)
encoder_layout.addWidget(self.read_encoder_button)

second_row_layout.addWidget(encoder_group)

# Third Row: Submit Parameters, Reset Parameters, Reset Antenna Position
third_row_layout = QHBoxLayout()
main_layout.addLayout(third_row_layout)

self.submit_button = QPushButton('Submit Parameters')
self.submit_button.clicked.connect(self.submit_parameters)
third_row_layout.addWidget(self.submit_button)

self.reset_button = QPushButton('Reset Parameters')
self.reset_button.clicked.connect(self.reset_all)
```

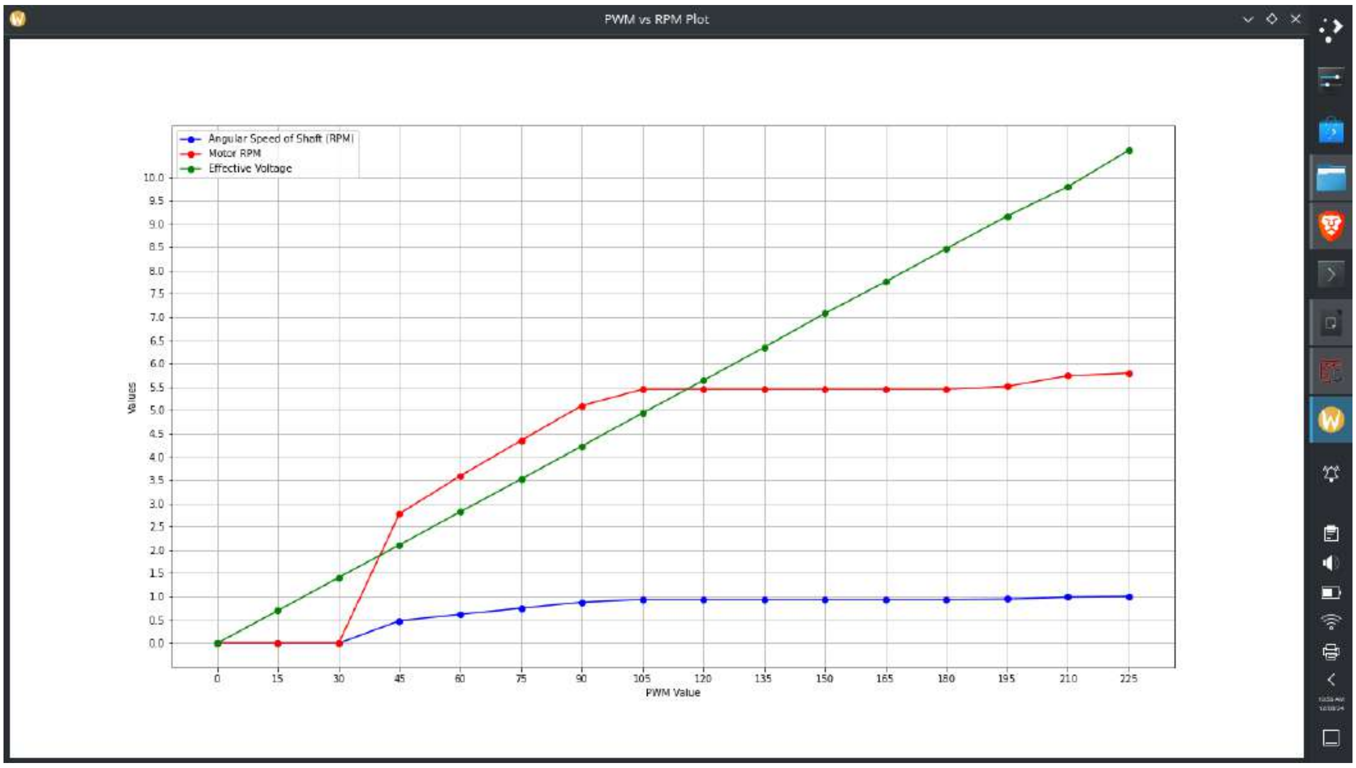
```
        third_row_layout.addWidget(self.reset_button)

        self.reset_antenna_button = QPushButton('Reset Antenna Position')
        self.reset_antenna_button.clicked.connect(self.reset_antenna_position)
        third_row_layout.addWidget(self.reset_antenna_button)

        # Spacer
        spacer = QSpacerItem(20, 40, QSizePolicy.Minimum, QSizePolicy.Expanding)
        main_layout.addItem(spacer)

        # Initialize settings from file
        self.read_motor_settings()

    # This will set the antenna position to encoder count of 0
    def reset_antenna_position(self):
        self.pulse_entry.clear()
        self.send_serial_command('R')
        QMessageBox.information(self, "Reset Antenna Position", "Antenna position has been reset.")
        self.log_action("Antenna position reset")

    # Read files from the encoder
    def read_motor_settings(self):
        if os.path.exists(file_path):
            with open(file_path, 'r') as file:
                lines = file.readlines()
                if len(lines) >= 2:
                    self.pulse_entry.setText(lines[1].strip())

    # Send pulses per revolution
    def submit_pulses(self):
        pulses_per_rev = self.pulses_entry.text().strip()
        self.send_serial_command(f"P:{pulses_per_rev}")
        self.log_action(f'Submitted pulses per revolution: {pulses_per_rev}')

    # Send the gear ratio
    def submit_gear_ratio(self):
        gear_ratio = self.gear_ratio_entry.text().strip()
        self.send_serial_command(f"G:{gear_ratio}")
        self.log_action(f'Submitted gear ratio: {gear_ratio}')

    # Send the shaft to antenna ratio
    def submit_shaft_ratio(self):
        shaft_ratio = self.shaft_ratio_entry.text().strip()
        self.send_serial_command(f"E:{shaft_ratio}")
        self.log_action(f'Submitted shaft to antenna ratio: {shaft_ratio}')

    # Send angle entered
    def submit_angle(self):
        angle = self.pulse_entry.text().strip()
        self.send_serial_command(f"A:{angle}")
        self.log_action(f'Submitted angle: {angle}')
```

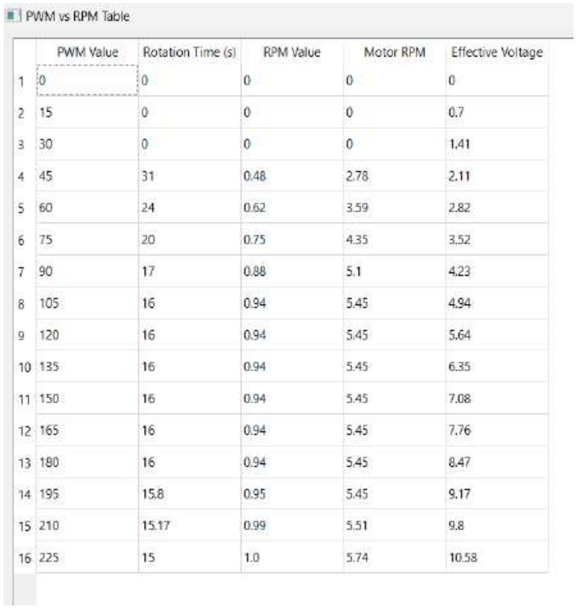
```
# Error handling for speed PWM
def set_speed(self):
    speed_text = self.speed_entry.text()
    try:
        speed = int(speed_text)
        if 0 <= speed <= 255:
            command = f'M:{speed}'  # Sending 'M' command with speed value to Arduino
            self.send_serial_command(command)
            self.speed_label.setText(f'Speed Set: {speed}')
            self.log_action(f'Set speed: {speed}')
        else:
            self.speed_label.setText('Error: Value out of range (0-255)')
            self.log_action('Speed set error: Value out of range (0-255)')
    except ValueError:
        self.speed_label.setText('Error: Invalid input. Please enter a number between 0 and 255.')
        self.log_action('Speed set error: Invalid input')

# Handle forward button click event
def forward(self):
    self.send_serial_command('F')
    QMessageBox.information(self, "Direction", "Motor is moving forward.")
    self.log_action("Motor moving forward")

# Handle backward button click event
def backward(self):
    self.send_serial_command('B')
    QMessageBox.information(self, "Direction", "Motor is moving backward.")
    self.log_action("Motor moving backward")

# Read encoder value
def read_encoder(self):
    self.send_serial_command('E')  # Command to read encoder value
    self.log_action("Read encoder value")

# Handle direction count button click event
def direction_count(self):
    self.send_serial_command('D')
    QMessageBox.information(self, "Direction Count", "Direction count command sent.")
    self.log_action("Direction count command sent")

# Reset all parameters to default
def reset_all(self):
    self.pulses_entry.clear()
    self.gear_ratio_entry.clear()
    self.shaft_ratio_entry.clear()
    self.pulse_entry.clear()
    self.speed_entry.clear()
    QMessageBox.information(self, "Reset", "All parameters have been reset.")
    self.log_action("All parameters reset")

# Handle settings submission
def submit_parameters(self):
        self.submit_pulses()
        self.submit_gear_ratio()
        self.submit_shaft_ratio()
        self.submit_angle()
        self.set_speed()
        self.log_action("Submitted all parameters")

    # Dark mode settings
    def setDarkMode(self):
        palette = QPalette()
        palette.setColor(QPalette.Window, QColor(53, 53, 53))
        palette.setColor(QPalette.WindowText, Qt.white)
        palette.setColor(QPalette.Base, QColor(25, 25, 25))
        palette.setColor(QPalette.AlternateBase, QColor(53, 53, 53))
        palette.setColor(QPalette.ToolTipBase, Qt.white)
        palette.setColor(QPalette.ToolTipText, Qt.white)
        palette.setColor(QPalette.Text, Qt.white)
        palette.setColor(QPalette.Button, QColor(53, 53, 53))
        palette.setColor(QPalette.ButtonText, Qt.white)
        palette.setColor(QPalette.BrightText, Qt.red)
        palette.setColor(QPalette.Highlight, QColor(142, 45, 197).lighter())
        palette.setColor(QPalette.HighlightedText, Qt.black)
        self.setPalette(palette)

    # Logging function
    def log_action(self, action):
        with open(log_file_path, 'a') as log_file:
            log_file.write(f'{datetime.datetime.now()}: {action}\n')

if __name__ == '__main__':
    app = QApplication(sys.argv)
    motor_control = None  # This should be replaced with the actual motor control object if used
    mainWin = MotorControlGUI(motor_control)
    mainWin.show()
    sys.exit(app.exec_())
```

10.2 Arduino Program:-

```
#include <Arduino.h> //default arduino library

const int enablePin = 9;    // PWM pin for motor speed control
const int int1Pin = 8;      // Motor driver input 1
const int int2Pin = 7;      // Motor driver input 2

int encoderPin1 = 2;    //Encoder pin 2, White
int encoderPin2 = 3;    //Encoder pin 3, Green

volatile long encoderValue = 0;
int pulsesPerRevolution = 0;        //encoder value that is PPM
long maximumEncoderValue = 0;       //maximum value the encoder can go while rotating the antenna
for 360 degree.
long requiredPulsesForRotation;       //pulses that are required to move antenna by requested angle

int motorRatio = 1;          // Default to 1 to prevent division by zero
int antennaAngle;       //store the antenna angle from python
int newPulses;        //store the ppm from python
int speed;        //store the value of speed that we get from arduino
int gearBoxRatio =1;
int encoderRatio = 1;

void setup() {
  Serial.begin(9600);   //serial communication baudrate setting

  //motor pins declared as output
  pinMode(enablePin, OUTPUT);       //pwm pin for motor
  pinMode(int1Pin, OUTPUT);       //direction pin 1
  pinMode(int2Pin, OUTPUT);       //direction pin 2

  //turn on the internal pullup
  pinMode(encoderPin1, INPUT_PULLUP);
  pinMode(encoderPin2, INPUT_PULLUP);

  //keep the encoder to change for detection in the change of value
  attachInterrupt(digitalPinToInterrupt(encoderPin1), updateEncoder, CHANGE);
  attachInterrupt(digitalPinToInterrupt(encoderPin2), updateEncoder, CHANGE);

}

 //converting the pulses to maximumEncoderValue we can get for 360 degree of rotation
void conversion(){
  long pulsesPerRevolution = newPulses * 4;       //multiply it by 4 to get encoder count
  gearBoxRatio = motorRatio / encoderRatio;
  maximumEncoderValue = (pulsesPerRevolution * gearBoxRatio);      //multiply it by gear ratio to get
maximum pulses antenna can rotate
  requiredPulsesForRotation= (maximumEncoderValue / 360)*antennaAngle;        //pulses the antenna
will rotate for
  Serial.println("Maximum Encoder Value is:- ");
  Serial.println(maximumEncoderValue);
  Serial.println("Required Pulses for Rotation is:- ");
  Serial.println(requiredPulsesForRotation);
}

void loop() {
 {
  Serial.println(encoderValue);
  delay(1000); //just here to slow down the output, and show it will work  even during a delay
}

  if (Serial.available() > 0) {
    char command = Serial.read();       //read commands that are being sent from python

    switch (command) {            //set the case as per the commands sent from the python gui
      case 'A':  // Set Antenna Angle
        antennaAngle = Serial.parseInt();
        Serial.print("Received Antenna Angle: ");
        Serial.println(antennaAngle);
        break;

      case 'P':  // Set Pulse Count per Revolution
        newPulses = Serial.parseInt();
        Serial.print("Pulse count per revolution: ");
        Serial.println(newPulses);
        break;

      case 'G':  // Set motor to shaft ratio
        gearRatio = Serial.parseInt();
        Serial.print("Gear Ratio: ");
        Serial.println(gearRatio);
        break;

      case 'E': // Set shaft to antenna ratio
        shaftRatio= Serial.parseInt();
        Serial.print("Shaft Ratio: ");
        Serial.println(shaftRatio);
        break;

      case 'F':  // clockwise direction
        forward(speed, requiredPulsesForRotation);
        Serial.println("Forward direction");
        conversion();
        break;

      case 'B':  // anticlockwise direction
        backward(speed, requiredPulsesForRotation);
        Serial.println("Backward direction");
        conversion();
        break;

      case 'D':  // Direction Check
        directionCheck(encoderValue);
        break;

      case 'R':  // Reset the antenna position to zero
        reset (requiredPulsesForRotation);
        break;

      case 'S':   // Stop the motor
        stopMotor();
        Serial.println("Motor has been stopped");
        break;

      case 'M':   //set speed i.e pwm in motor
        speed = Serial.parseInt();
        Serial.print("Speed Received : ");
        Serial.println(speed);
        break;

      default:      //handle unknown commands
        Serial.print("Unknown command: ");
        Serial.println(command);
        break;
    }
  }
}

void updateEncoder() {        //arduino counting using A and B pulse from the encoder  !!do not change!!
  static int lastEncoded = 0;
  int MSB = digitalRead(encoderPin1);
  int LSB = digitalRead(encoderPin2);
  int encoded = (MSB << 1) | LSB;
  int sum = (lastEncoded << 2) | encoded;

  if (sum == 0b1101 || sum == 0b0100 || sum == 0b0010 || sum == 0b1011)
    encoderValue++;
  if (sum == 0b1110 || sum == 0b0111 || sum == 0b0001 || sum == 0b1000)
    encoderValue--;

  lastEncoded = encoded;
}

// function that will rotate the motor clockwise
void forward(int speed, long requiredPulsesforRotation) {
  long targetEncoderValue = requiredPulsesforRotation;     //calculation for clockwise rotation of motor
  Serial.println("The target encoder value that is calculated is:- ");
  Serial.println(targetEncoderValue);

  while (encoderValue <= targetEncoderValue) {
    analogWrite(enablePin, speed);
    digitalWrite(int1Pin, HIGH);
    digitalWrite(int2Pin, LOW);
    Serial.println(encoderValue);
    encoderValue++;
  }
  stopMotor();  // stop after the motor completes the loop
  Serial.println("Motor has been stopped ");
}

//function that will rotate the motor anticlockwise
void backward(int speed, long requiredPulsesforRotation) {
  long targetEncoderValue = requiredPulsesforRotation;       //calculation for anticlockwise rotation of the
motor
  Serial.println("The target encoder value that is calculated is:- ");
  Serial.println(targetEncoderValue);

  while (encoderValue >= targetEncoderValue) {
    analogWrite(enablePin, speed);
    digitalWrite(int1Pin, LOW);
    digitalWrite(int2Pin, HIGH);
    Serial.println(encoderValue);
    encoderValue--;
  }
  stopMotor();    //stop after the motor completes the loop
  Serial.println("Motor has been Stopped ");
}

// function that will stop the motor
void stopMotor() {
  digitalWrite(int1Pin, LOW);
  digitalWrite(int2Pin, LOW);
  Serial.println("Function call of motor stopped");
}

//function to check rotation direction of the antenna
void directionCheck(long currentEncoderValue) {
  static long previousEncoderValue = 0;

  if (currentEncoderValue > previousEncoderValue) {
    Serial.println("Clockwise Direction");
  } else if (currentEncoderValue < previousEncoderValue) {
    Serial.println("Anti-Clockwise Direction");
  } else {
    Serial.println("No Change in Direction");
  }
  previousEncoderValue = currentEncoderValue;
}



// rotate the antenna to its reset position, which here is set to 0
void reset(long requiredPulsesForRotation) {
  while (encoderValue >= 0) {
    analogWrite(enablePin,255);
    digitalWrite(int1Pin, LOW);
    digitalWrite(int2Pin, HIGH);
    Serial.println("!!RESETTING THE ANTENNA POSITIONS!!!");
  }
  stopMotor();
  Serial.println("Parameters reset. Antenna returned to original position.");
}
```

CHAPTER 12
OBSERVATION WITH THE PROPOSED RFI ANTENNA CONTROL SYSTEM




PWM vs RPM Table

	PWM Value	Rotation Time (s)	RPM Value	Motor RPM	Effective Voltage
1	0	0	0	0	0
2	15	0	0	0	0.7
3	30	0	0	0	1.41
4	45	31	0.48	2.78	2.11
5	60	24	0.62	3.59	2.82
6	75	20	0.75	4.35	3.52
7	90	17	0.88	5.1	4.23
8	105	16	0.94	5.45	4.94
9	120	16	0.94	5.45	5.64
10	135	16	0.94	5.45	6.35
11	150	16	0.94	5.45	7.08
12	165	16	0.94	5.45	7.76
13	180	16	0.94	5.45	8.47
14	195	15.8	0.95	5.45	9.17
15	210	15.17	0.99	5.51	9.8
16	225	15	1.0	5.74	10.58

The table shows the relationship between the PWM value applied to a motor and the resulting RPM of the antenna shaft.

- **PWM Value:** This column represents the pulse width modulation value applied to the motor, which controls the motor's speed. Higher PWM values correspond to higher motor speeds.
- **Rotation Time (s):** This column shows the time it takes for the antenna shaft to complete 90 degrees of rotation. A lower rotation time means a faster RPM.
- **RPM Value:** This column represents the revolutions per minute of the antenna shaft, calculated based on the rotation time. Higher RPM values indicate faster rotation.

Observations:

- As the PWM value increases, the rotation time generally decreases, leading to a higher RPM.
- There is a non-linear relationship between the PWM value and the RPM, with larger increments in PWM resulting in smaller increases in RPM, especially at higher PWM values.
- The relationship between PWM and RPM is likely influenced by factors such as motor characteristics, load on the antenna shaft, and the efficiency of the motor.

Explanation:

The motor's speed is determined by the frequency and duty cycle of the PWM signal. A higher duty cycle (i.e., a larger PWM value) means more power is applied to the motor, resulting in faster rotation. However, the motor's mechanical characteristics and the load on the antenna shaft can affect the actual RPM achieved.

This data can be used to understand how PWM controls the motor speed and its impact on the antenna shaft's RPM. It can be used to optimize the PWM value to achieve the desired antenna rotation speed for various applications.

CHAPTER 13
CONCLUSION AND FUTURE SCOPE

In conclusion, this project has successfully achieved its objective of developing a highly precise and controllable antenna system for effective RF signal detection. Through the seamless integration of an Arduino microcontroller with a Python-based user interface, the system delivers precise antenna positioning based on user-defined parameters. The Arduino's efficient command processing and motor control, combined with the user-friendly Python interface, have ensured that the antenna can be accurately directed to capture desired RF signals.

The incorporation of feedback mechanisms, such as encoders, has been pivotal in maintaining the desired accuracy, allowing for real-time adjustments and ensuring consistent performance. This feedback loop has not only enhanced the precision of the antenna movements but also contributed to the system's robustness and reliability.

Additionally, the project's attention to error-handling and calibration routines has guaranteed that the system operates reliably under various environmental conditions, with minimal performance degradation. Effective power management has been implemented to support extended operational periods, ensuring the system's endurance and reliability.

Overall, this project stands as a testament to the successful fusion of hardware and software, achieving a sophisticated antenna control system capable of precise RF signal detection. The meticulous design, implementation, and testing have culminated in a robust, reliable, and highly accurate system that meets the project's ambitious goals.

Future Scope:

1. Implementing a Braking System:
- Introduce a braking system to mitigate motor backlash and enhance precision.

2. Replacing the DC Motor with a Stepper Motor:
- Use a stepper motor instead of a DC motor to achieve finer control over antenna movements.

3. Limiting Motor Rotation to 270 Degrees:
- Restrict the motor's rotation to 270 degrees to prevent damage to the wiring system.

4. Integrating an Absolute Encoder:
- Add an absolute encoder alongside the existing encoder to obtain precise positional data of the antenna.

5. Installing Limit Switches:
- Place limit switches at both ends to prevent the antenna from exceeding its operational limits.

6. Developing a Power Management Plan:
- Create a power management strategy to ensure the Arduino remains operational during power failures.

7. Establishing a Reset Position:
- Define a reset position for the antenna, so that in case of encoder failure, the antenna returns to a known parking position, providing a reliable reference point for subsequent operations.

CHAPTER 14
REFERENCES

1. Tata Institute of Fundamental Research. Handle/2301/321. n.d.
http://library.ncra.tifr.res.in:8080/jspui/handle/2301/321

2. Spyder Project. (n.d.). Spyder. GitHub. Retrieved August 6, 2024, from
https://github.com/spyder-ide/spyder

3. KDE (a.d.). Tutorials. KDE UserBase. Retrieved August 6, 2024, from
https://userbase.kde.org/Tutorials

4. Arduino. Retrieved August 6, 2024, from
https://www.arduino.cc/en/software

5. Debain. (n.d.). Debain tutorial. Retrieved August 6, 2024, from
https://www.debian.org/doc/manuals/debian-reference/ch01.en.html

6. Tata Institute of Fundamental Research. Servo System
http://www.ncra.tifr.res.in/ncra/gmrt/sub-systems/servo-system-for-gmrt-antennas