RFI IDENTIFICATION AND STATISTICAL ANALYSIS USING HISTORICAL DATA

Submitted on successful completion of the project , on account of the Students training program (STP) - 2025

Under the guidance of

Mr. Ankur
&
Prof.Divya Oberoi

Submitted by

ANJANA BALAKUMARAN R

FRONT END DEPARTMENT , RFI LAB

GIANT METREWAVE RADIO TELESCOPE

KHODAD

AUGUST 2025

BONAFIDE CERTIFICATE

This is to certify that this project titled **" RFI IDENTIFICATION AND STATISTICAL ANALYSIS USING HISTORICAL DATA"** submitted to **Giant Metrewave Radio Telescope , Khodad,** is a bonafide record of work done by **Anjana Balakumaran R,** under my supervision at the **Giant Metrewave Radio Telescope , NCRA - TIFR** from "**5th May 2025 to 1st August 2025 ".**

Mr.Ankur
Engineer E, RFI & Front End department ,
GMRT, NCRA-TIFR

Prof. Divya Oberoi
Reader -F,
NCRA - TIFR

Place : GMRT , Khodad

Date : 01/08/2025

DECLARATION BY AUTHOR

I hereby declare that the work presented in this report titled **"RFI identification and statistical analysis using historical data"** is my own and I further declare that:This report has not been submitted previously, either in part or full, for the award of any degree, diploma, or other qualification.The work is original and does not contain any plagiarised content.All sources of information have been appropriately cited and acknowledged.

Anjana Balakumaran R

II - Msc physics

PSGR Krishnammal college for women

Date : 01/08/2025

Place : Giant Metrewave Radio Telescope - Khodad .

ACKNOWLEDGEMENT

I would like to express my sincere gratitude to all those who supported and guided me throughout the course of this project.

First and foremost , I would like to express my sincere gratitude to **Prof. Yashwant Gupta**, Director of GMRT, for providing me the opportunity to undertake this project at such a prestigious facility. I am equally thankful to **Prof. Ishwara Chandra**, Dean of GMRT. Their leadership and commitment to scientific research have been a constant source of inspiration, and I am truly honoured to have been a part of the vibrant research environment at GMRT.

I would like to thank, **Mr. Ankur**, Engineer – E, Front End Lab, Giant Metrewave Radio Telescope (GMRT) , for his valuable guidance, consistent support, and insightful suggestions as my project guide. His technical expertise and encouragement were crucial to the successful completion of this work, and I personally thank him for providing me the opportunity to work in a professional research environment .

I am also deeply grateful to **Prof. Divya Oberoi**, Reader – F, NCRA–TIFR, for his constant motivation, academic mentorship, and for serving as my co-guide. His feedback helped me improve the quality and depth of this project.

I would also like to extend my heartfelt thanks to **Shri. Sureshkumar**, Engineer – G and Group Head, Front End Lab, GMRT, for his encouragement throughout the duration of my project.

Lastly, I appreciate the entire team at front end lab, GMRT for creating a supportive and intellectually stimulating atmosphere during my time here.

ABSTRACT

Radio Frequency Interference (RFI) poses a significant challenge in radio astronomy, corrupting observational data and obscuring weak cosmic signals. This project presents a robust, data-driven pipeline for the detection, visualisation, and temporal analysis of RFI across multiple antennas, frequency bands, and time ranges. The core of the detection methodology employs **Huber loss-based polynomial baseline fitting**, which offers resilience against strong outliers while preserving baseline integrity. RFI is flagged by identifying deviations exceeding a dynamic **3σ threshold** above this fitted baseline.

The pipeline integrates multiple visualisation modules: **waterfall plots** to display spectral evolution over time, **bar graphs** showing band-wise occupancy statistics, and **calendar heat maps** to monitor daily observation durations and RFI time percentages. It supports flexible user inputs for antenna selection, channel, frequency band, and date range. Additional modules compute **frequency vs. time occupancy percentages** and track the degradation of detection sensitivity over the years due to increasing RFI, as reflected in rising sigma thresholds.

To counteract this degradation, the system supports referencing **archival baselines from cleaner years** for consistent RFI detection in present-day data. Overall, this modular, scalable framework enhances the ability to monitor and mitigate RFI contamination in large-scale radio observatories.

TABLE OF CONTENTS

1.INTRODUCTION

1.1 Institution profile

NCRA has setup a unique facility for radio astronomical research using the metre wavelengths range of the radio spectrum , known as the Giant Metrewave Radio Telescope (GMRT), located at a site about 80km north of Pune. GMRT consists of fully steerable gigantic parabolic dishes of 45 metre diameter , each spread over distances of upto 25km. GMRT is one of the most challenging experimental programmes in basic sciences undertaken by Indian scientists and engineers .

The number and configuration of the dishes were optimised to meet the principal astrophysical objectives which require sensitivity at high angular resolution as well as ability to image radio emissions from diffuse extended regions . Fourteen of the thirty dishes are located more or less randomly in a compact Central Array in a region of about 1 sq km. The remaining sixteen dishes are spread out along the 3 arms of an approximately Y-shaped configuration over a much larger region , with the longest interferometric baseline about 25 km.

The multiplication or correlation of radio signals from all the 435 possible pairs of the antennas or interferometers over several hours will thus enable radio images of celestial objects to be synthesised with a resolution equivalent to that obtainable with a single gigantic dish 25 kilometre diameter ! The array will operate in six frequency bands centred around 50, 153, 233, 325, 610 and 1420 MHz. All these feeds provide dual polarisation outputs . In some configuration , dual frequency observations are also possible .

The higher angular resolution achievable will range from about 60 arcsec at the lowest frequencies to about 2 arcsec at 1.4GHz.

GMRT an indigenous project . The construction of 30 large dishes at a relatively small cost has been possible due to an important technological breakthrough achieved by Indian scientists and engineers in the design of the light weight , low cost dishes . The design is based on what is being called the ‘SMART’ concept - for Stretch Mesh Attached to Rope Trusses.

The dish has been made light weight and of low solidity by replacing the conventional backup structure by series of rope trusses stretched between 16 parabolic frames made of tubular steel . The wire ropes are tensioned suitably to make mosaic of plane facets approximating a parabolic surface .

A light-weight thin wire mesh (made of 0.55 mm diameter stainless steel wire) with a grid size varying from 10 X 10 mm in the central part of the dish to 20 X 20 mm in the outer parts, stretched over the rope truss facets forms the reflecting surface of the dish. The low-solidity design cuts down the wind forces by a large factor and is particularly suited to Indian conditions where there is no snowfall in the plains. The overall wind forces and the resulting torques for a 45-m GMRT dish are similar to those for only a 22-m dish of conventional design, thus resulting in substantial savings in cost.

The dish is connected to a `cradle' which is supported by two elevation bearings on a yoke placed on a 3.6 m diameter slewing-ring bearing secured on the top of a 15 metre high concrete tower. The weight of the disk is about 80 tonnes and the counter-weight is about 40 tonnes. The dishes have alt-azimuth mount. The salient parameters and specifications of each dish are summarised in the Table.

1.2 What is RFI?

Radio Frequency Interference (RFI) is the disruption or contamination of a desired radio signal by the presence of unwanted electromagnetic waves. These interfering signals overlap with the frequency bands used for legitimate communication or observation, leading to a degradation of signal quality, data loss, or even complete failure of radio-based systems. RFI is particularly problematic in fields that rely on the detection of weak signals, such as radio astronomy, remote sensing, satellite communications, and radar systems. In such applications, even a faint interfering signal can overwhelm a legitimate one, rendering the data unusable or corrupt.

RFI can originate from a wide range of sources and is broadly categorised into natural and artificial (man-made) interference. Natural RFI sources include lightning discharges, auroras, cosmic background noise, and solar flares, which emit broadband radio noise that can interfere with observations. However, the vast majority of problematic RFI in modern environments comes from man-made sources. These include telecommunications infrastructure such as mobile towers, FM/TV broadcast stations, Wi-Fi networks, satellite constellations, and radar systems. Additionally, many common electronic devices—computers, LED lights, power lines, switching power supplies, car engines, and even electric toothbrushes—emit unintended radio signals that contribute to the growing RFI environment.

The nature of RFI can vary—it may be narrowband or broadband, stationary or drifting, continuous or intermittent. Narrowband RFI appears at specific frequencies and can resemble legitimate signals, making detection challenging. Broadband RFI spans a wide frequency range and can significantly contaminate large portions of the spectrum. Time-varying RFI, such as radar pulses or intermittent transmissions, poses additional challenges due to its unpredictable nature.

In radio astronomy, RFI is particularly damaging because the astronomical signals being observed are typically many orders of magnitude weaker than man-made transmissions. As a result, RFI can completely obscure the cosmic signals or introduce false detections that mimic real astronomical phenomena. To address this, observatories employ a multi-layered approach to RFI mitigation. This includes physical measures such as placing telescopes in remote, radio-quiet zones, using shielded and filtered electronics, and aligning with regulatory protections that restrict emissions in certain bands. On the software side, advanced signal processing techniques are used to identify and suppress RFI. These methods include statistical outlier detection, robust polynomial or spline fitting to subtract baselines, time-frequency masking, machine learning classifiers, and algorithms like Huber loss fitting that are robust to outliers.

The increasing density of wireless communication infrastructure, the expansion of satellite constellations (e.g., Starlink), and the general electrification of modern life have led to a significant rise in RFI levels worldwide. This trend threatens not only the quality of scientific research but also the reliability of communications and navigation systems. To combat this, international and national regulatory bodies such as the International Telecommunication Union (ITU), the Federal Communications Commission (FCC), and local spectrum agencies define rules for frequency allocation, emission limits, and usage rights to help minimise interference. However, enforcement can be difficult, especially in regions lacking infrastructure or legal control.

In summary, Radio Frequency Interference is a growing and complex challenge that affects a wide range of technologies and scientific fields. Understanding its sources, characteristics, and mitigation strategies is essential for ensuring the integrity and reliability of systems that depend on clean radio frequency environments. As humanity becomes increasingly dependent on wireless communication, the importance of effective RFI management will only continue to grow.

1.3 Threats posed by RFI to Radio astronomy :

Radio Frequency Interference (RFI) poses serious and growing threats to radio astronomy, a field that relies on the detection of extremely faint natural radio signals from celestial sources. Since astronomical radio emissions are often billions of times weaker than man-made transmissions, even minimal RFI can drown them out, leading to several major problems:

1. **Loss of Data Integrity**:
   RFI can contaminate observational data by introducing artificial signals that are indistinguishable from true cosmic sources. This leads to corrupted datasets where important astrophysical information may be hidden, distorted, or completely erased.

2. **False Detections**:
   RFI can mimic genuine astronomical signals, resulting in false positives. For example, a pulsed signal from a radar or satellite may resemble a pulsar or fast radio burst (FRB), misleading researchers and causing wasted effort and incorrect scientific conclusions.

3. **Reduced Sensitivity**:
   Continuous or broadband RFI raises the noise floor in a telescope's receiver system, reducing its sensitivity. This means that weaker astronomical signals that would otherwise be detectable get lost in the increased background noise.

4. **Loss of Observing Time**:
   Interference often forces astronomers to discard portions of their data or reschedule observations entirely. In some cases, entire frequency bands become unusable for science, limiting what researchers can study and reducing the value of expensive observing time on national or international facilities.

5. **Spectral Band Pollution**:
   Certain frequency bands are internationally reserved for radio astronomy (e.g., the hydrogen line at 1.42 GHz). However, spillover emissions from adjacent bands or unauthorised transmissions can pollute these protected zones, degrading critical observations that cannot be done at other frequencies.

6. **Increased Cost and Complexity**:
   Mitigating RFI requires sophisticated signal processing, robust filtering techniques, and careful site planning—all of which increase the complexity and cost of radio astronomy projects. Additionally, observatories often have to invest in RFI monitoring and

mitigation infrastructure to keep their data usable.

7. **Threat to Future Discoveries**:
   As RFI continues to grow with the expansion of wireless technologies and satellite constellations, the window for making groundbreaking discoveries in radio astronomy is narrowing. If left unmanaged, RFI could prevent the detection of extremely faint or rare signals—such as those from the early universe, exoplanet atmospheres, or extraterrestrial intelligence—that are crucial to advancing our understanding of the cosmos.

8. **Cumulative Global Impact**:
   Because radio waves are not confined by national borders, RFI from one region can affect observatories far away, especially in space-based radio telescopes or in global Very Long Baseline Interferometry (VLBI) networks. The cumulative interference from multiple sources across the globe threatens the collaborative nature of modern radio astronomy.

In essence, RFI endangers the core capability of radio astronomy: the ability to observe the universe through its faintest signals. Without strong protection, regulation, and technological innovation to combat interference, the future of this observational science—and the discoveries it can make—may be significantly constrained.

2. STATISTICAL ANALYSIS

The below mentioned methodologies are some of the techniques which can be used to create a descriptive analysis on the RFI stats .

A. Correlation Studies

Correlation studies aim to uncover relationships between observed RFI events and external environmental or operational factors that may be influencing them. By analysing timestamped RFI data alongside logs from weather systems, device operation records, solar activity indices, or even regional power consumption logs, one can identify patterns suggesting that certain external conditions are conducive to RFI occurrences. For instance, a spike in RFI may consistently coincide with local thunderstorms, indicating a strong weather-based origin. Similarly, solar flares or geomagnetic storms can emit radio waves that interfere with ground-based radio telescopes. These correlations not only help in attributing causes to the interference but also in developing mitigation strategies, such as avoiding observations during high solar activity or creating exclusion masks around certain time windows or events. Statistically, correlation coefficients, heat maps, or multivariate regression models may be employed to quantify these associations, offering insights into which factors most strongly contribute to RFI contamination.

B. Outlier Detection

Outlier detection focuses on identifying rare, extreme, or anomalous RFI events that deviate significantly from normal background levels. These high-impact events may be caused by unexpected and powerful sources such as radar systems, transient satellite transmissions, or short bursts from faulty electronic equipment. From a data analytics standpoint, these outliers are typically spikes in amplitude or power levels that stand out in frequency-time heat maps. In Python, library such as `scipy` is used to implement peak-finding algorithms, which detect these sharp deviations from the baseline. Statistically, methods like the Z-score, interquartile range (IQR), or Huber loss-based residual thresholding can also be applied to filter out or flag outliers. Detecting these events is crucial for data cleaning and for avoiding the misinterpretation of spurious interference as genuine astronomical signals. It also helps in building databases of known RFI signatures, which can later be excluded or corrected using automated pipelines.

C. Time Series Analysis

Time series analysis is used to examine how RFI evolves over time, looking for recurring patterns, trends, or periodic behaviours. By treating RFI data as a function of time, one can apply statistical models to identify whether certain types of interference are intermittent or persistent. Intermittent RFI is characterised by periodic bursts that appear at regular intervals—perhaps due to scheduled equipment cycles, recurring satellite passes, or human activity patterns. On the other hand, persistent RFI represents continuous interference over extended durations, often due to infrastructure like broadcasting towers or continuous electronics emissions. Through techniques such as autocorrelation, seasonal decomposition, or Fourier transforms, one can distinguish between these two types. Identifying time-dependent structures in RFI helps astronomers plan observation windows more effectively, as well as construct filters or masks that align with known RFI periodicities. This method also aids in long-term monitoring of site conditions and in verifying whether regulatory measures (like spectrum restrictions) are effective over time.

D. Predictive Modelling

Predictive modelling in the context of RFI involves using historical data to forecast when and where interference is likely to occur in the future. By analysing past RFI events, their frequencies, amplitudes, and time distributions, machine learning or statistical algorithms can be trained to predict similar events in upcoming observation cycles. For example, if a specific frequency band showed consistent RFI patterns over the past year during a particular time of day, the model can forecast similar disruptions in the present or near future. This is especially useful in observation planning, where astronomers might choose to avoid certain frequency bands or reschedule sessions based on RFI forecasts. Moreover, predictive models can leverage the shape or profile of past RFI-contaminated bands to interpret present-day data, particularly if the current signal is ambiguous. Techniques like ARIMA (AutoRegressive Integrated Moving Average), LSTM (Long Short-Term Memory neural networks), or random forest regressors are often used to build these models. Overall, predictive modelling enables proactive mitigation rather than reactive cleanup, making it an essential component of modern RFI management strategies.

3. DATA FILE AND USER INPUT SPECIFICATIONS

3.1 Data file details :

- The data used in this project is sourced from the 60:1 monitoring tool.
- The file sizes range approximately from 1 MB to 150 MB.
- These data files contain time-stamped amplitude values that correspond to various spectral channels across four frequency bands—band 2, band 3, band 4, and band 5.
- The dataset includes measurements from a total of 30 antennas, each with two polarisations, resulting in 60 distinct channels.
- Each channel consists of data recorded over 401 frequency steps, providing detailed frequency resolution for analysis.

3.2 User input specifications :

The project interface is designed to accept specific user inputs that tailor the data analysis process according to user-defined criteria.

- The key input features include the selection of the frequency band, with available options being band 2, band 3, band 4, and band 5.
- Users are also required to specify the antenna name, which ranges covers all the antenna names such as C00, C01, up to W06.
- In addition to the antenna, the user selects the channel name, either CH1 or CH2, corresponding to the two available polarisations.
- Finally, users define the time window of interest by providing a start and stop date, which helps in filtering and loading only the relevant data files for the selected observation period.

4. GRAY SCALE PLOT / WATERFALL PLOT

As part of the visual analysis in this project, grayscale plots were generated to represent the variation in signal amplitude over both frequency and time. These plots are often referred to as **waterfall plots**, and they serve as a foundational visualisation tool in the detection and analysis of Radio Frequency Interference (RFI). Each grayscale plot is a 2D matrix where the **x-axis corresponds to frequency**, the **y-axis corresponds to time (or individual data files arranged chronologically)**, and the **pixel intensity represents the signal amplitude** at each frequency-time point.

Lighter shades in the plot typically indicate **higher amplitudes**, which may suggest the presence of strong or anomalous signals, while darker regions represent **lower, background-level signals** that are closer to the system noise floor.

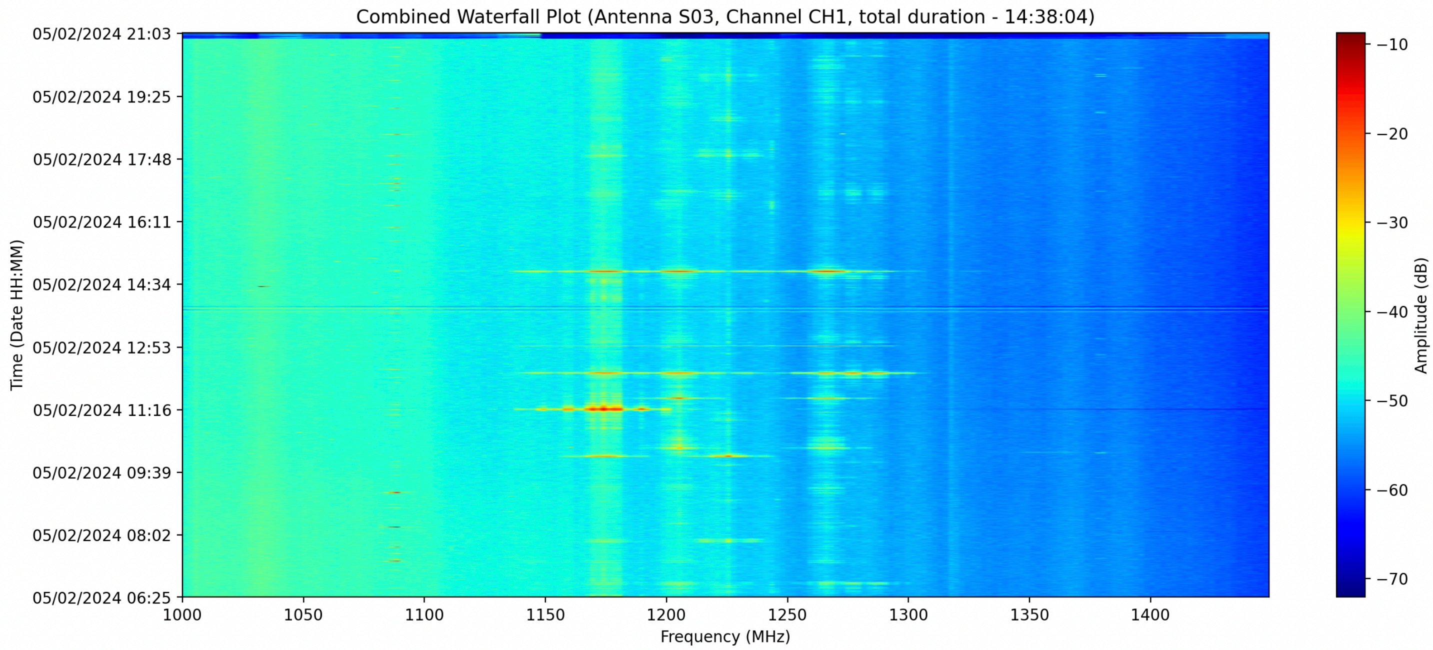

figure :4.1 : Gray scale plot for band 5 data file , dated : 05/02/2024, for antenna :S03, channel :CH1.

These plots allow us to visually track the presence, persistence, and frequency spread of RFI events. For example, **horizontal streaks** in the plot indicate **persistent interference at a fixed frequency over time**, which is characteristic of continuous RFI sources like broadcasting towers or industrial equipment. In contrast, **vertical streaks** or bands of lighter intensity may suggest **broadband events** affecting a large range of frequencies simultaneously, possibly caused by transient environmental noise or certain types of satellite transmissions.

Intermittent RFI, which appears sporadically at fixed intervals, shows up as periodic light patches along the same frequency row. By examining these patterns, users can infer not only the existence but also the temporal behaviour of RFI sources—whether they are continuous, intermittent, sudden, or fading. The use of a **grayscale colour map** rather than a rainbow or multi-coloured scale is intentional and beneficial for subtle pattern recognition. Grayscale preserves the focus on contrast without introducing perceptual biases that colour maps may cause. It also facilitates easier quantitative interpretation when comparing across multiple plots, especially when amplitude values are normalised or thresholded for consistency.

In this project, these grayscale waterfall plots were generated for specific combinations of **bands, antennas, channels, and date ranges**, as chosen by the user inputs. This modularity allowed for high flexibility in inspection—enabling users to zoom in on specific antennas or channels affected by RFI in particular frequency bands. These plots served as a first step in identifying problematic regions of the spectrum, guiding further stages of the analysis such as baseline fitting, thresholding, outlier detection, and statistical modelling.

4.1 Waterfall plot for band -2 :

- The grayscale plot generated for Band 2 (100 MHz to 300 MHz) displays the variation in signal amplitude across 401 frequency steps over the specified observation period.
- This plot corresponds to the selected antenna and channel, and is constrained within the start and stop date provided by the user.
- Band 2 typically covers lower frequency ranges, where man-made interference from local communication systems or environmental noise sources is more likely to occur.
- In the plot, RFI sources manifest as bright vertical lines at specific frequency bins, indicating interference. In some regions, faint and sporadic bright patches suggest intermittent activity, which could be linked to cyclical or time-dependent emitters such as nearby industrial machinery or scheduled transmissions.
- The plot helps in distinguishing stable spectral regions from highly contaminated ones, which is critical for planning clean observation windows in Band 2.

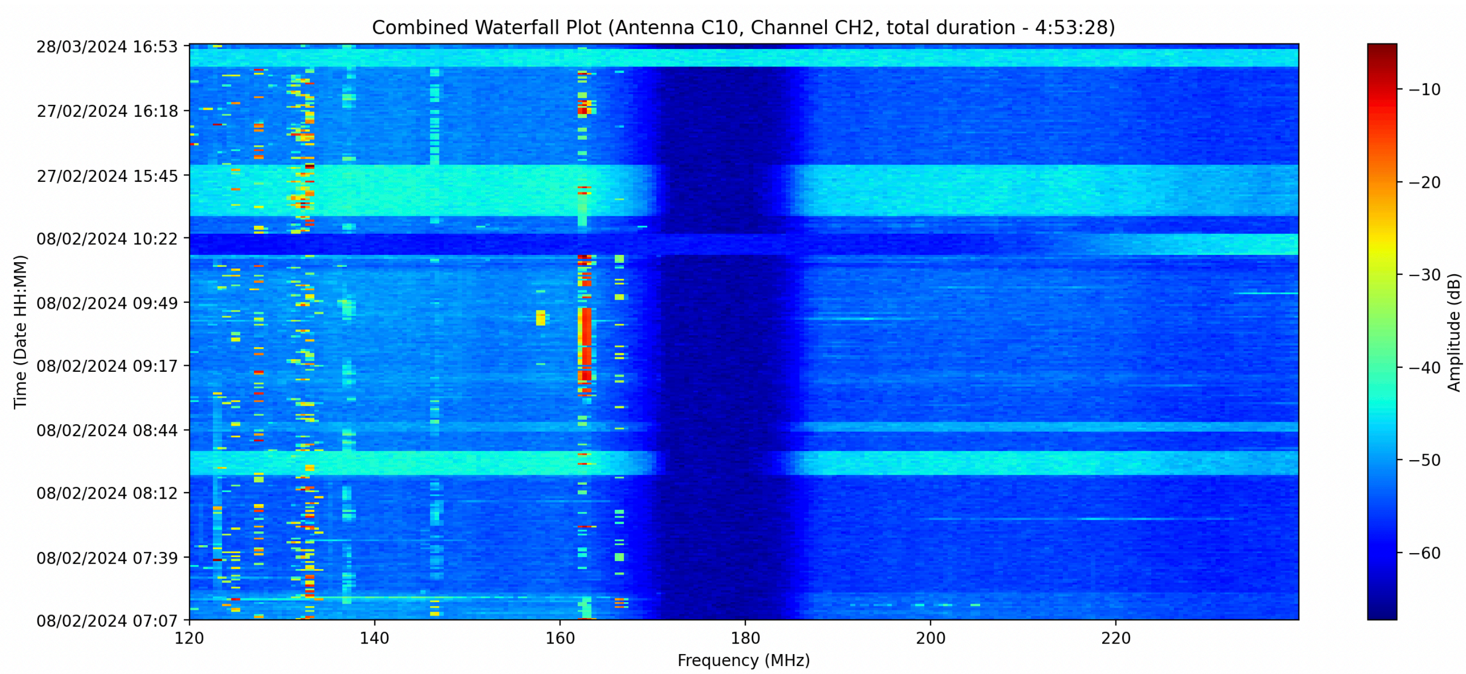

figure :4.2 : Gray scale plot for band 2 data file , dated : 08/02/2024 - 28/03/2024, for antenna :C10, channel :CH2.

4.2 Waterfall plot for band -3 :

- For Band 3, the grayscale plot offers a detailed representation of signal amplitudes detected over time, again filtered by user-specified antenna, channel, and observation period.

- Band 3 often encompasses mid-range frequencies, ranging from 175 MHz to 575 MHz

- The grayscale intensity patterns show several regions of moderate to strong RFI, with some bright frequency bins persisting over a significant portion of the time axis.

- This may indicate fixed-location transmitters or equipment operating in nearby environments.

- In contrast, some frequency bands remain consistently dark, suggesting clean and usable regions of the spectrum.

- This plot is particularly useful for identifying frequency ranges in Band 3 that could be masked out or flagged in downstream analysis.

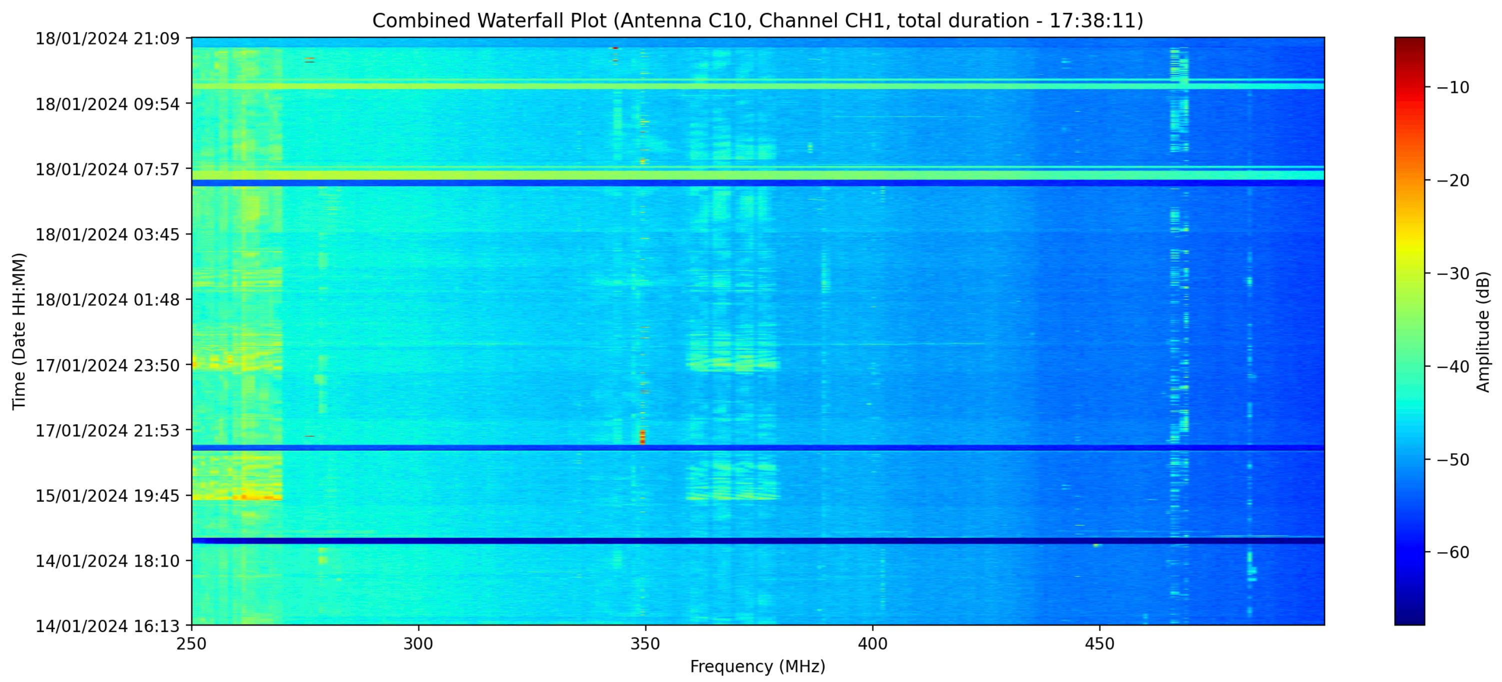

figure :4.3 : Gray scale plot for band 3 data file , dated : 14/01/2024 - 18/01/2024, for antenna :C10, channel :CH1.

4.3 Waterfall plot for band - 4 :

- The Band 4 grayscale plot focuses on slightly higher frequency regions, 500 MHz to 1000MHz again constrained to the chosen antenna, channel, and observation time window.

- Compared to lower bands, RFI in Band 4 can appear more structured or bursty, often originating from transient sources such as mobile networks.

- The grayscale plot highlights these phenomena through sharp, high-intensity peaks that are temporally narrow and localised in frequency.

- This makes Band 4 particularly interesting for time-sensitive interference detection. In some cases, band edges may show increased activity, which can be due to spillover from adjacent transmission bands.

- This visualisation provides valuable insight into which parts of Band 4 remain stable and which are vulnerable to unpredictable interference bursts.

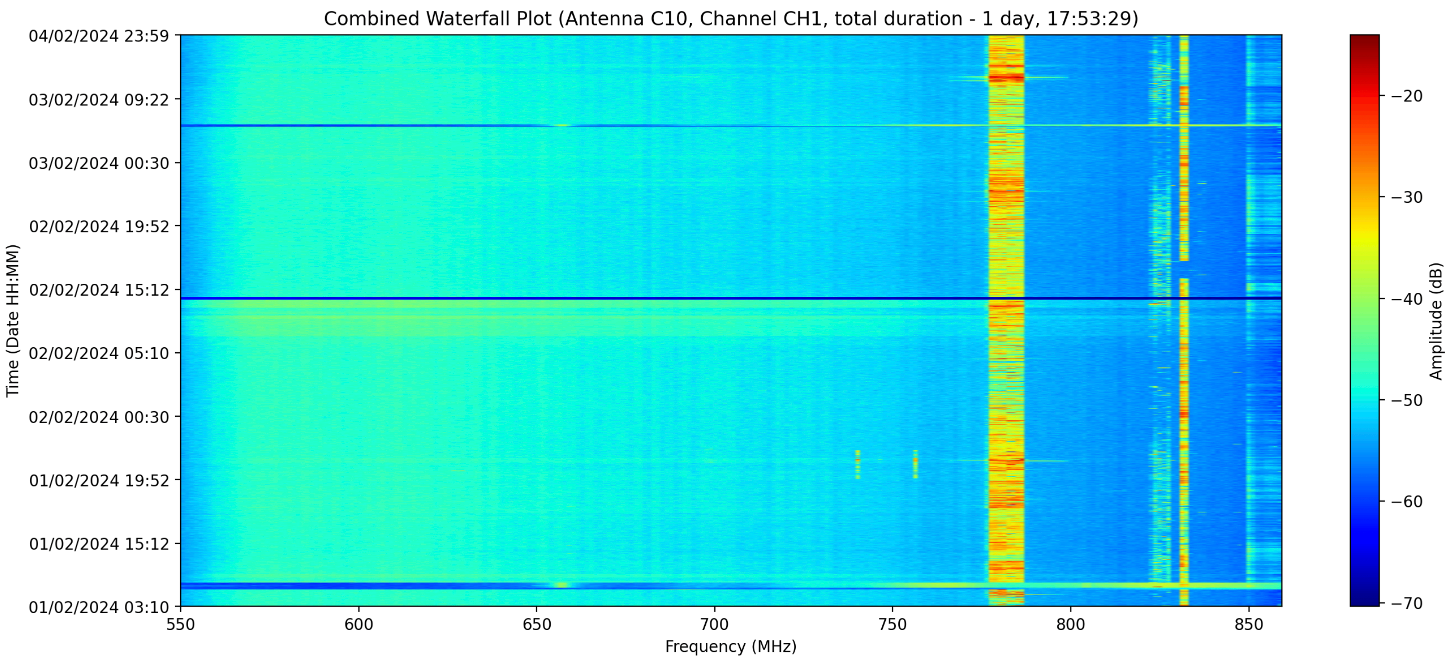

figure :4.4 : Gray scale plot for band 4 data file , dated : 01/02/2024 - 04/02/2024 , for antenna :C10, channel :CH1.

4.4 Waterfall plot for band - 5 :

- The grayscale plot for Band 5 reveals signal behaviour in the higher frequency end of the observational spectrum, 800MHz to 17000Mhz.
- Band 5 is often susceptible to interference from satellite constellations, airborne sources, and high-frequency broadband equipment.
- The plot for the selected antenna and channel across the defined date range shows a mixture of low-level background activity and high-impact transient RFI events.
- These appear as scattered bright regions or clusters, which can be irregular in both frequency and time, making them more challenging to model.
- However, some persistent frequency bands can still be identified, often corresponding to predictable emitters.
- The Band 5 plot plays a crucial role in assessing the dynamic nature of RFI in high-frequency bands and helps refine frequency filters applied in subsequent processing.

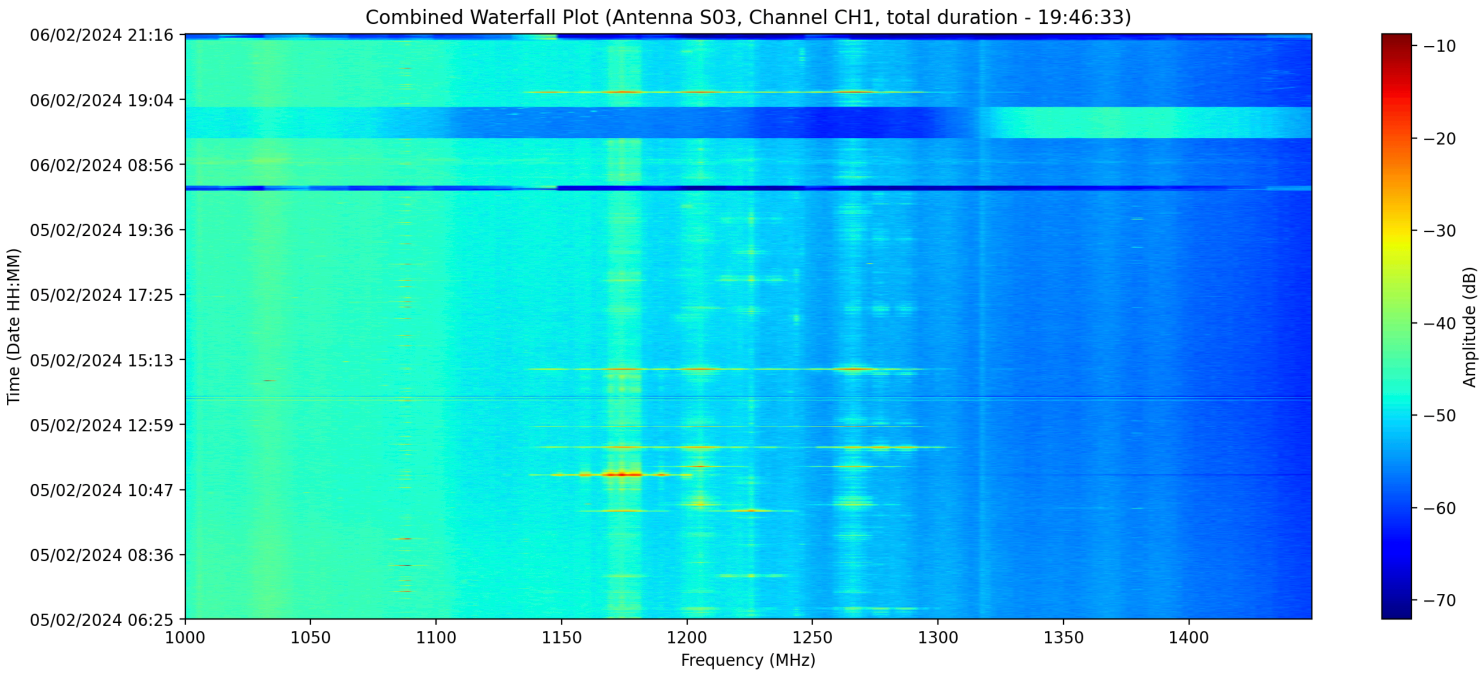

figure :4.5 : Gray scale plot for band 5 data file , dated : 05/02/2024 - 06/02/2024, for antenna :S03, channel :CH1.

4.5 Python Code :

```
import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

from datetime import datetime, timedelta

import os


# Frequency band options for user input

band_frequencies = {

    "band 2": {"start": 100, "stop": 300},

    "band 3": {"start": 175, "stop": 575},

    "band 4": {"start": 500, "stop": 1000},

    "band 5": {"start": 800, "stop": 1700},

    "all bands": {"start": 0, "stop": 1500},

}


# Subranges of bands used specifically for calculation (excludes the shoulder of the band shape )

calculation_band_frequencies = {

    "band 2": {"start": 120, "stop": 240},

    "band 3": {"start": 250, "stop": 500},

    "band 4": {"start": 550, "stop": 860},

    "band 5": {"start": 1000, "stop": 1450},

    "all bands": {"start": 120, "stop": 1450},

}


# Get valid band input from user

while True:
```

```
    user_band_input = input('Enter the band name (available bands: Band 2, Band 3, Band 4, Band 5, all bands): ').lower()


    if user_band_input in band_frequencies:
        selected_band_info = band_frequencies[user_band_input]
        calculation_band_info = calculation_band_frequencies[user_band_input]

        # Extract start/stop frequency values for plotting and calculation
        start_frequency_in_MHz = selected_band_info["start"]
        stop_frequency_in_MHz = selected_band_info["stop"]
        calc_start_freq_mhz = calculation_band_info["start"]
        calc_stop_freq_mhz = calculation_band_info["stop"]

        print(f"Selected {user_band_input}: Start Frequency = {start_frequency_in_MHz} MHz, Stop Frequency = {stop_frequency_in_MHz} MHz")
        break
    else:
        print("Invalid band name. Please enter one of the bands mentioned here(Band 2, Band 3, Band 4, Band 5).")

# Converts to Hz for later comparison with filenames
start_frequency = start_frequency_in_MHz * 1000000
stop_frequency = stop_frequency_in_MHz * 1000000

# Defines number of bins and generate frequency array for plotting
num_freq_bins = 401
calc_freq_increment = (calc_stop_freq_mhz - calc_start_freq_mhz) / num_freq_bins
frequencies = np.array([calc_start_freq_mhz + i * calc_freq_increment for i in range(num_freq_bins)])
```

```
# Gets the antenna and channel input
antenna_name = input("enter the antenna name:  ").upper()
channel_name = input("enter the channel name: ").upper()

# Validates date inputs and convert to datetime.date
while True:
    try:
        start_date_str = input("Enter the start date (DD/MM/YYYY): ")
        stop_date_str = input("Enter the stop date (DD/MM/YYYY): ")

        user_start_date = datetime.strptime(start_date_str, "%d/%m/%Y").date()
        user_stop_date = datetime.strptime(stop_date_str, "%d/%m/%Y").date()

        if user_start_date > user_stop_date:
            print("Error: Start date cannot be after stop date. Please re-enter dates.")
        else:
            break
    except ValueError:
        print("Invalid date format. Please use DD/MM/YYYY.")

filename_array = []

# Scans current directory for matching files
for file_name in os.listdir('.'):
    if file_name.endswith('Hz.txt') and '_' in file_name:
        try:
```

```
    sections = file_name.split('_')

    if len(sections) < 5:
        print(f"  - Skipping malformed filename (not enough sections): {file_name}")
        continue

    # Extract dates from filename
    file_day = sections[0].strip()
    file_month = sections[1].strip()
    file_year = sections[2].strip()
    file_date_str = f"{file_day}/{file_month}/{file_year}"
    file_date = datetime.strptime(file_date_str, "%d/%m/%Y").date()

    # Checks if file date and frequency match user input
    if user_start_date <= file_date <= user_stop_date:
        start_freq_section = sections[3].strip()
        stop_freq_section = sections[4].replace('Hz.txt', '').strip()
        file_start_freq = float(start_freq_section)
        file_stop_freq = float(stop_freq_section)

        if file_start_freq == start_frequency and file_stop_freq == stop_frequency:
            filename_array.append(file_name)
            print(f"  - Found relevant file: {file_name}")

except (ValueError, IndexError) as e:
    print(f"  - Skipping file '{file_name}' due to parsing error: {e}")
    continue
```

```
filenames = filename_array

# Exits if no matching files found
if not filenames:
    print("No files found matching the specified date and frequency range.")
    exit()
else:
    print(f"Successfully identified {len(filenames)} files for processing.")

# Function to load and filter data from a file based on antenna and channel
def load_data(filename):
    try:
        df = pd.read_csv(filename, header=None, sep='\s+')
    except FileNotFoundError:
        print(f" file not found :{filename}")
        exit()

    # Filter rows matching the selected antenna and channel
    filtered = df[(df[2] == antenna_name) & (df[3] == channel_name)].reset_index(drop=True)
    if filtered.empty:
        print(f'No data found for antenna {antenna_name} and channel {channel_name} in
{filename}')
        return [], np.array([])

    # Extract timestamp strings and amplitude matrix
    time_strings = filtered[0].astype(str)+' '+filtered[1].astype(str)
```

```
    amplitude_matrix = filtered.iloc[:,13:].astype(float).values

    # Apply frequency mask for calculation range
    full_freq_increment = (stop_frequency_in_MHz - start_frequency_in_MHz) / num_freq_bins
    full_freqs = np.array([start_frequency_in_MHz + i * full_freq_increment for i in
range(num_freq_bins)])
    freq_mask = (full_freqs >= calc_start_freq_mhz) & (full_freqs <= calc_stop_freq_mhz)
    amplitude_matrix = amplitude_matrix[:, freq_mask]

    return time_strings.tolist(), amplitude_matrix

# Load data from all valid files
all_time_strings = []
all_amplitudes = []

for filename in filenames:
    current_time_strings, current_amplitudes = load_data(filename)
    all_time_strings.extend(current_time_strings)
    if current_amplitudes.size > 0:
        all_amplitudes.append(current_amplitudes)

# Exit if no valid data collected
if not all_time_strings:
    print("No data loaded from any of the files. Exiting.")
    exit()

# Convert timestamps to datetime objects and calculate seconds since midnight
```

```
combined_times = all_time_strings

combined_datetimes = [datetime.strptime(ts, "%d/%m/%Y %H:%M:%S") for ts in combined_times]

seconds_since_midnight = np.array([dt.hour * 3600 + dt.minute * 60 + dt.second for dt in combined_datetimes])


if not all_amplitudes:

    print("No amplitude data found after filtering. Exiting.")

    exit()


# Helper function to load all timestamps from a file (used for duration calculation)

def load_timestamps_all_rows(filename):

    try:

        df = pd.read_csv(filename, header=None, sep=r'\s+')

    except FileNotFoundError:

        print(f"File not found: {filename}")

        return []


    time_strings = df[0].astype(str) + ' ' + df[1].astype(str)

    try:

        return [datetime.strptime(t, "%d/%m/%Y %H:%M:%S") for t in time_strings]

    except Exception as e:

        print(f"Failed to parse timestamps in {filename}: {e}")

        return []


print("\n--- File Observation Durations (using >2s segmentation) ---")

total_duration = timedelta(0)
```

```
# Loop over files and compute segmented durations (gaps >2s split segments)
for file in filenames:
    timestamps = load_timestamps_all_rows(file)
    if len(timestamps) < 2:
        print(f"{file}: Not enough timestamps for duration calculation.")
        continue

    timestamps.sort()
    file_duration = timedelta(0)
    segment_start = timestamps[0]

    for i in range(1, len(timestamps)):
        delta = timestamps[i] - timestamps[i - 1]
        if delta > timedelta(seconds=2):
            file_duration += timestamps[i - 1] - segment_start
            segment_start = timestamps[i]

    file_duration += timestamps[-1] - segment_start
    print(f"{file}: duration = {file_duration}")
    total_duration += file_duration

# Stack all amplitude matrices vertically (combine all time steps)
combined_amplitudes = np.vstack(all_amplitudes)
total_rows = combined_amplitudes.shape[0]

# Sort data by datetime to prepare for plotting
sorted_indices = np.argsort(combined_datetimes)
```

```
combined_datetimes_sorted = np.array(combined_datetimes)[sorted_indices]

amplitudes_sorted = combined_amplitudes[sorted_indices]

seconds_since_midnight_sorted = seconds_since_midnight[sorted_indices]


# Generate waterfall plot

plt.figure(figsize=(14, 6))

plt.imshow(

    amplitudes_sorted,

    aspect='auto',

    extent=[frequencies[0], frequencies[-1], 0, len(combined_datetimes_sorted)],

    origin='lower',

    cmap='jet'

)


plt.colorbar(label='Amplitude (dB)')

plt.xlabel('Frequency (MHz)')

plt.ylabel('Time (Date HH:MM)')

plt.title(f'Combined Waterfall Plot (Antenna {antenna_name}, Channel {channel_name}, total
duration - {total_duration})')


# Set y-axis ticks to show readable date/time labels

num_ticks = 10

tick_indices = np.linspace(0, len(combined_datetimes_sorted) - 1, num_ticks, dtype=int)

tick_labels = [combined_datetimes_sorted[idx].strftime("%d/%m/%Y %H:%M") for idx in
tick_indices]

plt.yticks(tick_indices, tick_labels)

plt.tight_layout()

plt.show()
```

5.BAR GRAPH

5.1 what is a bar graph ?

The bar graph is used to provide a clear and comparative visual representation of total observation durations across different frequency bands. It plots **frequency bands along the x-axis** and the **corresponding total duration of data availability or observation time in hours along the y-axis**. Each bar represents the amount of time for which data has been recorded in that particular band, calculated based on user-specified start and stop durations of interest.

This visualisation is especially useful for understanding the data coverage across bands, identifying which bands have been more extensively observed, and highlighting any potential imbalances in observation time. For instance, a significantly shorter bar for a particular band might indicate fewer recordings, which could be due to limited availability, system downtime, or less scientific interest during that interval.

Conversely, taller bars suggest frequent or long-duration observations. By summarising the time spent observing each band in a compact and interpretable format, the bar graph aids in making informed decisions about band-wise analysis, data reliability, and future observation planning.

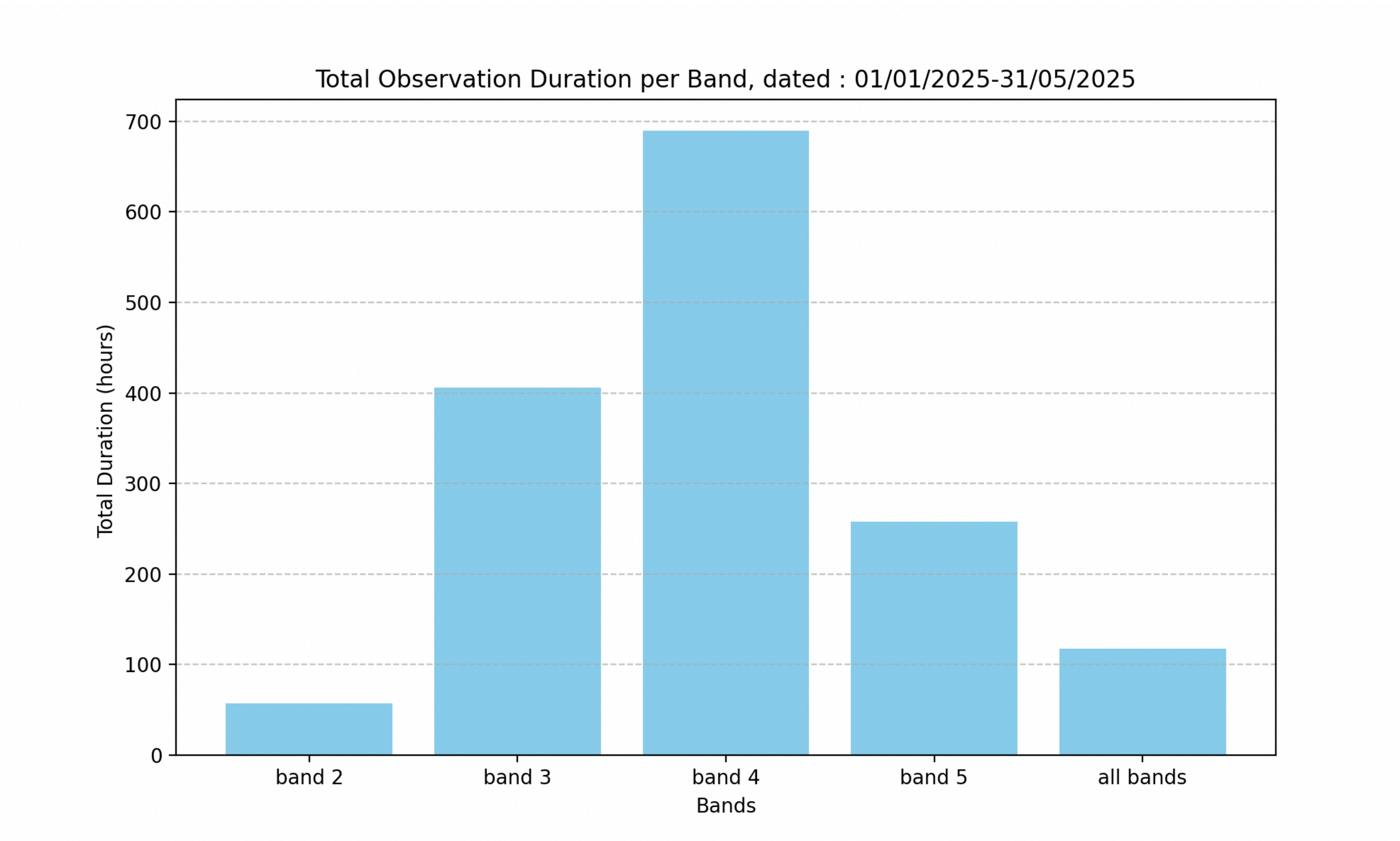

Figure 5.1: bar graph for all bands for a duration of 5 months, 01/01/2025 - 31/05/2025

The bar graph presented here (figure 5.1) illustrates the total observation duration for each frequency band over a fixed five-month window, spanning from **01/01/2025 to 31/05/2025**. This plot has been generated based on user-defined start and stop dates, and it visualises the cumulative observation time for **Band 2**, **Band 3**, **Band 4**, **Band 5**, and **all bands**. The x-axis of the graph represents the different frequency bands, while the y-axis denotes the total observation duration in **hours**.

From the graph, it is evident that **Band 4** had the highest observation duration during this period, exceeding **680 hours**, indicating either more scheduled observations or greater availability of usable data in this band. **Band 3** follows closely with over **400 hours** of observation. **Band 5** recorded a moderate duration of approximately **260 hours**, whereas **Band 2** had the least observation time, with under **60 hours**, possibly due to fewer data captures or limited availability. The bar labeled **"all bands"** shows the combined total duration where multi-band files were included, summing up to about **120 hours**.

This type of visualisation helps quantify the distribution of observational effort across frequency bands and is particularly useful for identifying under-observed bands, optimising scheduling, and comparing data density over time.

5.2 Python Code :

```
import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

from datetime import datetime, timedelta

import os


# Defines the frequency ranges (in MHz) for each band

band_frequencies = {

    "band 2": {"start": 100, "stop": 300},

    "band 3": {"start": 175, "stop": 575},

    "band 4": {"start": 500, "stop": 1000},

    "band 5": {"start": 800, "stop": 1700},

    "all bands": {"start": 0, "stop": 1500},

}


# Calculates frequency bins for each band (though not used later in this code)

for band, freq_range in band_frequencies.items():

    start_hz = freq_range["start"]

    stop_hz = freq_range["stop"]


    start_freq_mhz = start_hz * 1000000

    stop_freq_mhz = stop_hz * 1000000


    num_freq_bins = 401

    freq_increment = (stop_freq_mhz - start_freq_mhz) / num_freq_bins

    frequencies = np.array([start_freq_mhz + i * freq_increment for i in range(num_freq_bins)])


# Takes valid start and stop dates from the user
```

```
while True:
    try:
        start_date_str = input("Enter the start date (DD/MM/YYYY): ")
        stop_date_str = input("Enter the stop date (DD/MM/YYYY): ")

        user_start_date = datetime.strptime(start_date_str, "%d/%m/%Y").date()
        user_stop_date = datetime.strptime(stop_date_str, "%d/%m/%Y").date()

        if user_start_date > user_stop_date:
            print("Error: Start date cannot be after stop date. Please re-enter dates.")
        else:
            break

    except ValueError:
        print("Invalid date format. Please use DD/MM/YYYY.")

filename_array = []

# Selects files that match the date and frequency range
for file_name in os.listdir('.'):
    if file_name.endswith('Hz.txt') and '_' in file_name:
        try:
            sections = file_name.split('_')

            if len(sections) < 5:
                print(f"  - Skipping malformed filename (not enough sections): {file_name}")
                continue

            # Extracts the date from the filename
```

```
file_day = sections[0].strip()
file_month = sections[1].strip()
file_year = sections[2].strip()

file_date_str = f"{file_day}/{file_month}/{file_year}"
file_date = datetime.strptime(file_date_str, "%d/%m/%Y").date()

# Skips files outside the date range
if not (user_start_date <= file_date <= user_stop_date):
    continue

# Extracts start and stop frequency from the filename
start_freq_section = sections[3].strip()
stop_freq_section = sections[4].replace('Hz.txt', '').strip()
file_start_freq = float(start_freq_section)
file_stop_freq = float(stop_freq_section)

# Checks if the file overlaps with any of the defined bands
overlaps_any_band = False
for band, freq_range in band_frequencies.items():
    band_start = freq_range["start"] * 1e6
    band_stop = freq_range["stop"] * 1e6

    if file_stop_freq >= band_start and file_start_freq <= band_stop:
        overlaps_any_band = True
        break

# Appends valid file to the list
if overlaps_any_band:
```

```
            filename_array.append(file_name)
            print(f"  - Found relevant file: {file_name}")

        except (ValueError, IndexError) as e:
            print(f"  - Skipping file '{file_name}' due to parsing error: {e}")
            continue

filenames = filename_array

# Exits if no matching files were found
if not filenames:
    print("No files found matching the specified date and frequency range.")
    exit()
else:
    print(f"Successfully identified {len(filenames)} files for processing.")

# Loads a file and returns its DataFrame and combined timestamp strings
def load_data(filename):
    df = pd.read_csv(filename, header=None, sep='\s+')
    if df.shape[1] < 2:
        print(f"  - Skipping {filename}: not enough columns")
    time_strings = df[0].astype(str) + ' ' + df[1].astype(str)
    return df, time_strings

all_time_strings = []

# Loads timestamp strings from all selected files
for filename in filenames:
    df, time_strings = load_data(filename)
```

```
    all_time_strings.extend(time_strings)


# Exits if no timestamps were collected
if not all_time_strings:
    print("No data loaded from any of the files. Exiting.")
    exit()


file_observation_ranges = {}


# Initializes a dictionary to store total durations per band
band_durations = {band: timedelta(0) for band in band_frequencies}


# Calculates total observation duration for each band
for filename in filenames:
    df, current_file_time_strings = load_data(filename)


    if not current_file_time_strings.empty:
        # Converts timestamp strings to datetime objects and drops invalid entries
        cleaned_times = pd.to_datetime(current_file_time_strings, format="%d/%m/%Y %H:%M:
%S", errors='coerce')
        cleaned_times = cleaned_times.dropna()
        current_file_datetimes = cleaned_times.to_list()


        if not current_file_datetimes:
            continue


        # Calculates total observation time with segmentation based on >2 second gaps
        current_file_datetimes.sort()
        segment_start = current_file_datetimes[0]
```

```
        total_duration = timedelta(0)

        for i in range(1, len(current_file_datetimes)):
            delta = current_file_datetimes[i] - current_file_datetimes[i - 1]
            if delta > timedelta(seconds=2):
                total_duration += current_file_datetimes[i - 1] - segment_start
                segment_start = current_file_datetimes[i]

        total_duration += current_file_datetimes[-1] - segment_start

        # Extracts the frequency range from filename again
        sections = filename.split('_')
        file_start_freq = float(sections[3].strip())
        file_stop_freq = float(sections[4].replace('Hz.txt', '').strip())

        # Adds the duration to the matching band
        for band, freq_range in band_frequencies.items():
            band_start = freq_range["start"] * 1e6
            band_stop = freq_range["stop"] * 1e6

            if file_stop_freq == band_stop and file_start_freq == band_start:
                band_durations[band] += total_duration

# Prints total observation durations per band
print("\n--- Total Observation Durations by Band ---")
for band, duration in band_durations.items():
    print(f"{band.title()}: {duration}")

# Converts durations to hours for plotting
```

```
band_names = list(band_durations.keys())
durations_in_hours = [duration.total_seconds() / 3600 for duration in band_durations.values()]


# Generates a bar chart showing total duration per band
plt.figure(figsize=(10, 6))
plt.bar(band_names, durations_in_hours, color='skyblue')


plt.xlabel('Bands')
plt.ylabel('Total Duration (hours) ')
plt.title(f'Total Observation Duration per Band, dated : {start_date_str}-{stop_date_str}')
plt.grid(axis='y', linestyle='--', alpha=0.7)
plt.show()
```

6. CALENDAR PLOT

6.1 What is a calendar plot ?

The calendar heat map plot is a visual representation of the **daily observation duration** across a full calendar year, displayed in a month-by-day grid format. Each cell in the plot corresponds to a specific day of the year and is colour-coded based on the **number of observation hours recorded** on that day.

Darker shades (toward green) represent **higher durations**, while lighter shades (toward yellow or white) indicate **lower or no observation activity**. This plot provides a compact and intuitive way to identify patterns in observation coverage over time.

The **x-axis** of the heat map denotes the **day of the month**, while the **y-axis** lists the **months** from January to December. A colour bar is included alongside the plot to indicate the scale of observation duration in hours.

This format allows for easy detection of gaps in data collection, peak periods of observation activity, or seasonal trends in system availability or usage. In this project, the calendar heat map is generated **band-wise**, allowing users to focus on specific frequency bands individually and assess their temporal data coverage.

Such plots are particularly helpful for verifying the consistency and completeness of data over long time spans and for identifying periods of potential downtime, maintenance, or missing data. They also provide insight into operational efficiency and are valuable for planning future observations by highlighting underutilised periods.

6.2 Band wise calendar plot :

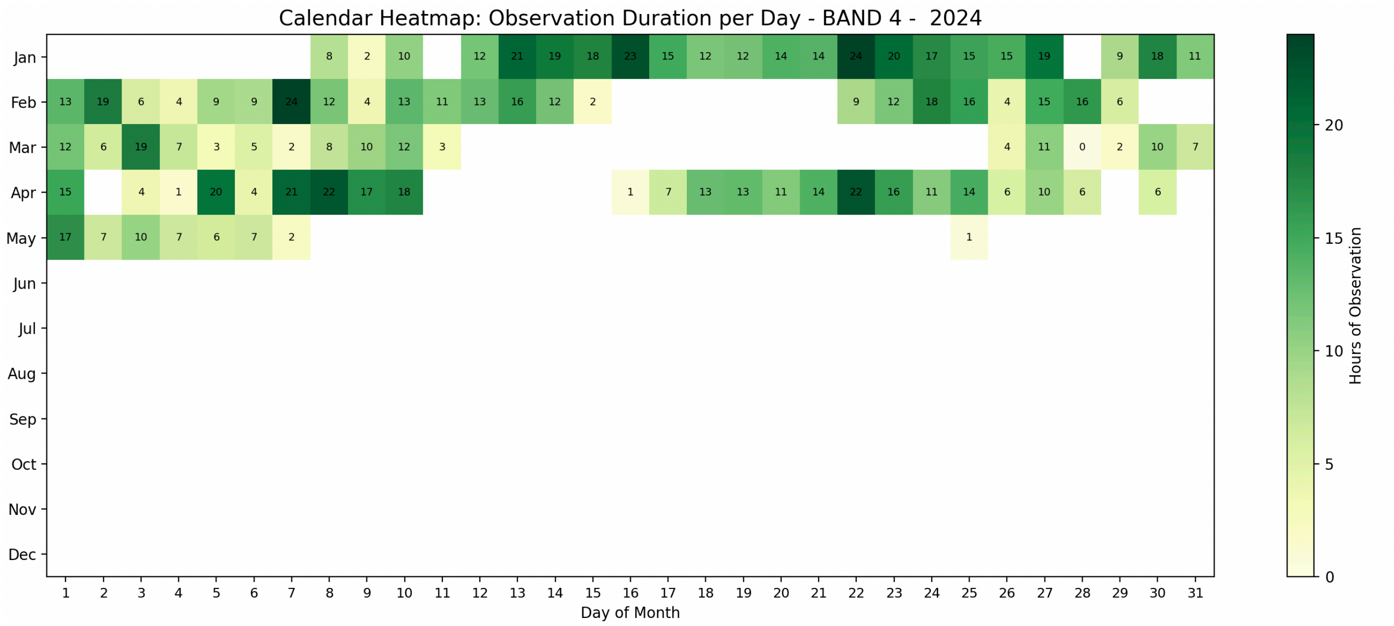

Figure 6.1 : Calendar plot for band -4 , for the duration of 5 months (01/01/2024 - 31/05/2024)

The calendar heat map shown above (figure 6.1) displays the **daily observation duration (in hours)** for **Band 4** across the year **2024**. Each cell corresponds to a specific calendar day and is colour-coded to reflect the total number of hours of observation recorded for that day. The **darker green shades** represent **higher observation durations**, while the **lighter shades and blank cells** indicate **lower or no observation activity**. The accompanying colour bar to the right provides a reference for interpreting the colour scale, ranging from 0 to over 20 hours per day.

From the heat map, it is evident that the majority of observation activity occurred between **January and May**, with no data available from **June to December**, as shown by the completely empty rows for those months. Notable high-observation days include **February 9 (24 hours)**, **January 20–23 (ranging from 18 to 23 hours)**, and **April 20–22**, which also recorded over 20 hours. These concentrated periods of activity suggest targeted or sustained observation campaigns during those days.

On the other hand, there are many days—particularly scattered in March and April—where only 1–4 hours were recorded, likely due to partial system usage or environmental interruptions.This heat map provides an effective summary of the time distribution of observations across the year and helps identify periods of high data availability versus operational gaps.It confirms that **Band 4** was most actively observed during the first five months of 2024, with **February and April showing the most consistent observation patterns**. The visualisation is valuable for both validating data completeness and informing the planning of future observations.

6.3 Python code :

```
# Import required libraries for numerical computation, data handling, date-time operations, plotting, and file management

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

from datetime import datetime, timedelta

import os

import calendar


# Define start and stop frequencies (in MHz) for each observational band

band_frequencies = {

    "band 2": {"start": 100, "stop": 300},

    "band 3": {"start": 175, "stop": 575},

    "band 4": {"start": 500, "stop": 1000},

    "band 5": {"start": 800, "stop": 1700},

}


# Prompt user to select a valid band from the predefined list

while True:

    selected_band = input("Enter the band to process (e.g., band 2, band 3, etc.): ").strip().lower()

    if selected_band not in band_frequencies:
```

```
        print("Invalid band. Please choose from:", list(band_frequencies.keys()))
    else:
        break

# Prompt user to enter a valid start and stop date for file filtering
while True:
    try:
        start_date_str = input("Enter the start date (DD/MM/YYYY): ")
        stop_date_str = input("Enter the stop date (DD/MM/YYYY): ")
        user_start_date = datetime.strptime(start_date_str, "%d/%m/%Y").date()
        user_stop_date = datetime.strptime(stop_date_str, "%d/%m/%Y").date()
        if user_start_date > user_stop_date:
            print("Start date must be before stop date.")
        else:
            break
    except ValueError:
        print("Invalid date format. Please use DD/MM/YYYY.")

# Extract frequency range in Hz for the selected band
band_range = band_frequencies[selected_band]
band_start_freq = band_range["start"] * 1e6
```

```
band_stop_freq = band_range["stop"] * 1e6

# Initialize a list to hold filenames that match the selected band and date range
filenames = []

# Iterate over files in the current directory to identify matching observation files
for file in os.listdir('.'):
    if file.endswith('Hz.txt') and '_' in file:
        try:
            cleaned_filename = file.replace(" ", "")
            parts = cleaned_filename.split('_')
            if len(parts) < 5:
                continue
            file_date = datetime.strptime(f"{parts[0]}/{parts[1]}/{parts[2]}", "%d/%m/%Y").date()
            if not (user_start_date <= file_date <= user_stop_date):
                continue
            file_start_freq = float(parts[3])
            file_stop_freq = float(parts[4].replace("Hz.txt", ""))
            if file_start_freq == band_start_freq and file_stop_freq == band_stop_freq:
                filenames.append(file)
        except:
```

```
        continue

# Exit if no matching files are found; otherwise, display the number of matches
if not filenames:
    print("No matching files found for the selected band and date range.")
    exit()
else:
    print(f"Found {len(filenames)} files for {selected_band}.")

# Define a function to load and parse timestamps from a given file
def load_timestamps(filename):
    df = pd.read_csv(filename, sep='\s+', header=None)
    time_strings = df[0].astype(str) + ' ' + df[1].astype(str)
    return pd.to_datetime(time_strings, format="%d/%m/%Y %H:%M:%S", errors='coerce').dropna().tolist()

# Initialize a dictionary to store timestamps organized by date
timestamps_by_date = {}

# Populate the dictionary with timestamps grouped by file date
for file in filenames:
    try:
```

```
        cleaned_filename = file.replace(" ", "")
        parts = cleaned_filename.split('_')
        file_date = datetime.strptime(f"{parts[0]}/{parts[1]}/{parts[2]}", "%d/%m/%Y").date()
        times = load_timestamps(file)
        if file_date not in timestamps_by_date:
            timestamps_by_date[file_date] = []
        timestamps_by_date[file_date].extend(times)
    except:
        continue


# Initialize a dictionary to store total observation durations per date
date_durations = {}


# Calculate total observation time for each date, accounting for gaps over 2 seconds
for date, times in timestamps_by_date.items():
    if not times:
        continue
    times.sort()
    total_duration = timedelta(0)
    segment_start = times[0]
    for i in range(1, len(times)):
```

```
        delta = times[i] - times[i - 1]
        if delta > timedelta(seconds=2):
            total_duration += times[i - 1] - segment_start
            segment_start = times[i]
    total_duration += times[-1] - segment_start
    date_durations[date] = total_duration

# Convert durations from timedelta to hours for each date
hours_by_date = {d: td.total_seconds() / 3600 for d, td in date_durations.items()}

# Initialize a heatmap array of shape (12 months × 31 days), filled with NaNs
heatmap = np.full((12, 31), np.nan)

# Fill heatmap with hourly values at the correct (month, day) positions
for d, hours in hours_by_date.items():
    heatmap[d.month - 1, d.day - 1] = hours

# Create a figure and axis for the calendar-style heatmap plot
fig, ax = plt.subplots(figsize=(16, 6))

# Display heatmap using imshow with a green-yellow color map
```

```python
c = ax.imshow(heatmap, aspect='auto', cmap='YlGn', vmin=0, vmax=24)


# Set x-axis ticks as days (1–31) and y-axis ticks as month names
ax.set_xticks(np.arange(31))
ax.set_xticklabels(np.arange(1, 32), fontsize=9)
ax.set_yticks(np.arange(12))
ax.set_yticklabels(calendar.month_abbr[1:], fontsize=10)


# Add numeric hour values to each heatmap cell (if data is present)
for m in range(12):
    for d in range(31):
        val = heatmap[m, d]
        if not np.isnan(val):
            ax.text(d, m, f"{val:.0f}", ha='center', va='center', color='black', fontsize=7)


# Add title, axis labels, and colorbar to the plot
plt.title(f"Calendar Heatmap: Observation Duration per Day - {selected_band.upper()} ",
fontsize=14)
plt.xlabel("Day of Month")
plt.ylabel("Month")
cbar = plt.colorbar(c, ax=ax, orientation='vertical')
cbar.set_label("Hours of Observation")
```

```
# Adjust layout and display the plot

plt.tight_layout()

plt.show()
```

6.4 Total observation duration calendar plot :

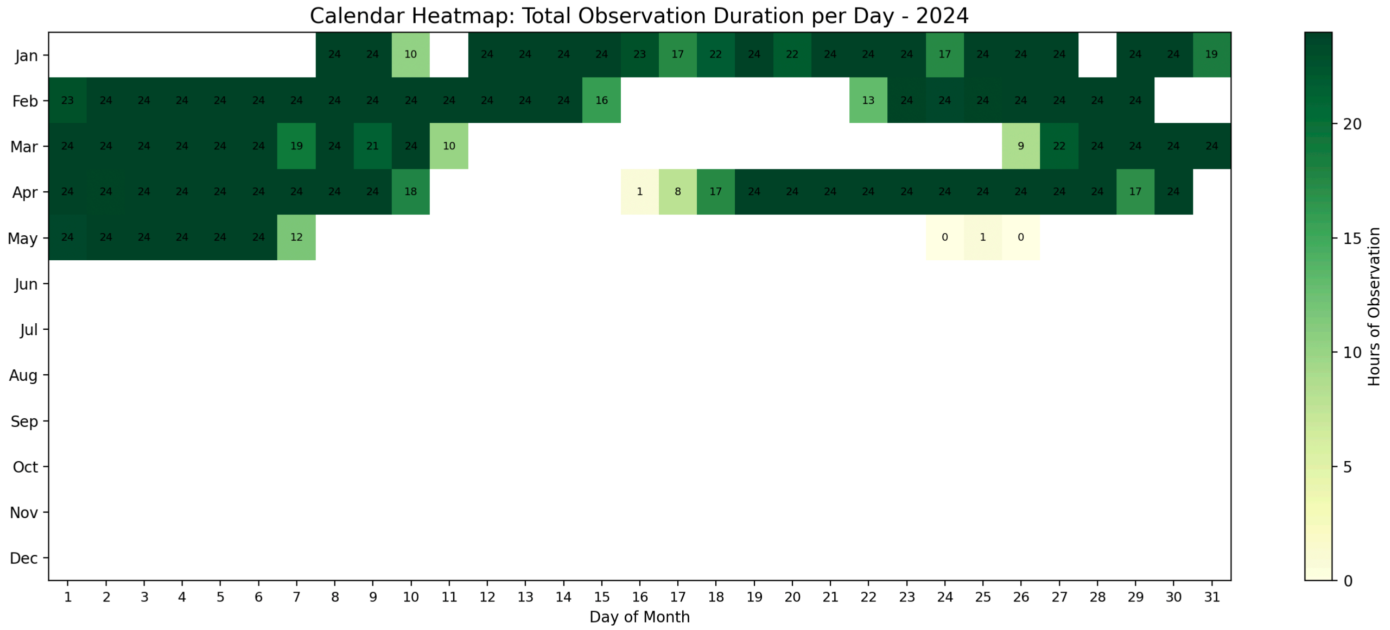

Figure 6.2 : Calendar plot for the duration of 5 months (01/01/2024 - 31/05/2024)

The above calendar heat map presents the **combined observation duration across all frequency bands**, displayed on a **day-by-day basis** for the year **2024**. Each cell in the plot represents a specific day of the month, with months listed along the y-axis and days along the x-axis. The intensity of the cell colour indicates the total number of observation hours recorded on that particular day, with darker shades of green corresponding to **higher observation durations**, and lighter or white shades representing **lower or no observations**. A colour bar on the right side of the plot maps the colour scale to numeric hour values.

The observation durations shown here is for a period of five months ,from **February through May**. Many days during this period show the **maximum possible 24 hours** of observation, suggesting full-day coverage. February and March, in particular, appear densely filled with high-duration values, indicating a phase of peak data acquisition. A few scattered days—such as **January 10**, **March 10**, and parts of late May—show relatively lower durations (ranging between 0 to 12 hours), possibly due to system downtime, maintenance, or external interference.

Overall, this plot provides a comprehensive visual summary of **temporal observation coverage** across the entire year and serves as a powerful diagnostic tool to assess operational consistency, highlight active and inactive periods, and validate the completeness of the dataset across all bands collectively.

6.5 Python Code :

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from datetime import datetime, timedelta
import os
import calendar

# Defines the frequency ranges (in MHz) for each band
band_frequencies = {
    "band 2": {"start": 100, "stop": 300},
    "band 3": {"start": 175, "stop": 575},
    "band 4": {"start": 500, "stop": 1000},
    "band 5": {"start": 800, "stop": 1700},
    "all bands": {"start": 0, "stop": 1500},
}

# Calculates frequency bins for each band (not reused later in the script)
for band, freq_range in band_frequencies.items():
    start_hz = freq_range["start"]
    stop_hz = freq_range["stop"]

    start_freq_mhz = start_hz * 1000000
    stop_freq_mhz = stop_hz * 1000000

    num_freq_bins = 401
    freq_increment = (stop_freq_mhz - start_freq_mhz)/num_freq_bins
    frequencies = np.array([start_freq_mhz + i * freq_increment for i in range(num_freq_bins)])

# Accepts user-specified start and stop dates
while True:
    try:
        start_date_str = input("Enter the start date (DD/MM/YYYY): ")
        stop_date_str = input("Enter the stop date (DD/MM/YYYY): ")

        user_start_date = datetime.strptime(start_date_str, "%d/%m/%Y").date()
        user_stop_date = datetime.strptime(stop_date_str, "%d/%m/%Y").date()

        if user_start_date > user_stop_date:
            print("Error: Start date cannot be after stop date. Please re-enter dates.")
        else:
            break
    except ValueError:
        print("Invalid date format. Please use DD/MM/YYYY.")

filename_array = []
```

```
# Selects files within the specified date range and overlapping frequency bands
for file_name in os.listdir('.'):
    if file_name.endswith('Hz.txt') and '_' in file_name:
        try:
            sections = file_name.split('_')

            if len(sections) < 5:
                print(f"  - Skipping malformed filename (not enough sections): {file_name}")
                continue

            file_day = sections[0].strip()
            file_month = sections[1].strip()
            file_year = sections[2].strip()

            file_date_str = f"{file_day}/{file_month}/{file_year}"
            file_date = datetime.strptime(file_date_str, "%d/%m/%Y").date()

            if not (user_start_date <= file_date <= user_stop_date):
                continue

            start_freq_section = sections[3].strip()
            stop_freq_section = sections[4].replace('Hz.txt', '').strip()
            file_start_freq = float(start_freq_section)
            file_stop_freq = float(stop_freq_section)

            overlaps_any_band = False
            for band, freq_range in band_frequencies.items():
                band_start = freq_range["start"] * 1e6
                band_stop = freq_range["stop"] * 1e6

                # Checks if file frequency overlaps with current band
                if file_stop_freq >= band_start and file_start_freq <= band_stop:
                    overlaps_any_band = True
                    break

            if overlaps_any_band:
                filename_array.append(file_name)
                print(f"  - Found relevant file: {file_name}")

        except (ValueError, IndexError) as e:
            print(f"  - Skipping file '{file_name}' due to parsing error: {e}")
            continue

filenames = filename_array

# Exits if no valid files found
if not filenames:
    print("No files found matching the specified date and frequency range.")
```

```
    exit()
else:
    print(f"Successfully identified {len(filenames)} files for processing.")

# Loads timestamp strings from a file
def load_data(filename):
    df = pd.read_csv(filename, header=None, sep='\\s+')
    if df.shape[1] < 2:
        print(f"  - Skipping {filename}: not enough columns")
    time_strings = df[0].astype(str)+' '+df[1].astype(str)
    return df, time_strings

all_time_strings = []

# Aggregates all timestamps across files
for filename in filenames:
    df, time_strings = load_data(filename)
    all_time_strings.extend(time_strings)

# Exits if no timestamps were found
if not all_time_strings:
    print("No data loaded from any of the files. Exiting.")
    exit()

band_durations = {band: timedelta(0) for band in band_frequencies}

# Calculates total observation duration per frequency band
for filename in filenames:
    df, current_file_time_strings = load_data(filename)

    if not current_file_time_strings.empty:
        cleaned_times = pd.to_datetime(current_file_time_strings, format="%d/%m/%Y %H:%M:
%S", errors='coerce')
        cleaned_times = cleaned_times.dropna()
        current_file_datetimes = cleaned_times.to_list()

        if not current_file_datetimes:
            continue

        start_time = min(current_file_datetimes)
        stop_time = max(current_file_datetimes)
        duration = stop_time - start_time

        sections = filename.split('_')
        file_start_freq = float(sections[3].strip())
        file_stop_freq = float(sections[4].replace('Hz.txt', '').strip())

        print(f"File: {filename}”).
```

```
        print(f"  Start time: {start_time.strftime('%d/%m/%Y %H:%M:%S')}")
        print(f"  Stop time:  {stop_time.strftime('%d/%m/%Y %H:%M:%S')}")
        print(f"  Duration:   {duration}")



        # Adds duration to the corresponding band
        for band, freq_range in band_frequencies.items():
            band_start = freq_range["start"] * 1e6
            band_stop = freq_range["stop"] * 1e6

            if file_stop_freq == band_stop and file_start_freq == band_start:
                band_durations[band] += duration

# Prints total observation duration per band
print("\n--- Total Observation Durations by Band ---")
for band, duration in band_durations.items():
    print(f"{band.title()}: {duration}")

# Loads datetime objects from a file

def load_data(filename):
    df = pd.read_csv(filename, header=None, sep='\\s+')
    time_strings = df[0].astype(str) + ' ' + df[1].astype(str)
    times = pd.to_datetime(time_strings, format="%d/%m/%Y %H:%M:%S", errors='coerce')
    return times.dropna().tolist()

# Organizes timestamps by date
timestamps_by_date = {}

for file_name in filenames:
    try:
        parts = file_name.strip().split('_')
        day, month, year = parts[0].strip(), parts[1].strip(), parts[2].strip()
        file_date_str = f"{day}/{month}/{year}"
        file_date = datetime.strptime(file_date_str, "%d/%m/%Y").date()

        times = load_data(file_name)
        if file_date not in timestamps_by_date:
            timestamps_by_date[file_date] = []
        timestamps_by_date[file_date].extend(times)
    except Exception as e:
        print(f"Error processing {file_name}: {e}")

# Calculates observation duration for each calendar date
date_durations = {}

for date, all_times in timestamps_by_date.items():
```

```
    if not all_times:
        continue
    all_times.sort()
    total_duration = timedelta(0)
    segment_start = all_times[0]

    for i in range(1, len(all_times)):
        delta = all_times[i] - all_times[i - 1]
        if delta <= timedelta(seconds=2):
            continue
        else:
            segment_end = all_times[i - 1]
            total_duration += segment_end - segment_start
            segment_start = all_times[i]

    total_duration += all_times[-1] - segment_start
    date_durations[date] = total_duration

# Prints total observation duration per calendar date
print("\n--- Total Observation Duration per Date  ---")
for date, duration in sorted(date_durations.items()):
    print(f"{date.strftime('%d/%m/%Y')}: {duration}")

# Converts durations to hours for heatmap
hours_by_date = {d: td.total_seconds() / 3600 for d, td in date_durations.items()}

# Initializes empty 12x31 heatmap matrix
heatmap = np.full((12, 31), np.nan)

# Fills heatmap matrix with observation durations (in hours)
for d, hours in hours_by_date.items():
    month_idx = d.month - 1
    day_idx = d.day - 1
    heatmap[month_idx, day_idx] = hours

# Plots the calendar heatmap
fig, ax = plt.subplots(figsize=(16, 6))
c = ax.imshow(heatmap, aspect='auto', cmap='YlGn', vmin=0, vmax=24)

ax.set_xticks(np.arange(31))
ax.set_xticklabels(np.arange(1, 32), fontsize=9)
ax.set_yticks(np.arange(12))
ax.set_yticklabels(calendar.month_abbr[1:], fontsize=10)

# Annotates each cell with hour values
for month in range(12):
    for day in range(31):
        hours = heatmap[month, day]
```

```
        if not np.isnan(hours):
            ax.text(day, month, f"{hours:.0f}", ha='center', va='center', color='black', fontsize=7)

plt.title("Calendar Heatmap: Total Observation Duration per Day - 2024", fontsize=14)
plt.xlabel("Day of Month")
plt.ylabel("Month")
cbar = plt.colorbar(c, ax=ax, orientation='vertical')
cbar.set_label("Hours of Observation")

plt.tight_layout()
plt.show()
```

7. HUBER LOSS

7.1 What is Huber Loss?

Huber loss is a **robust error function** used in regression tasks that is specifically designed to handle the presence of **outliers** in the data. Unlike traditional least squares regression, which penalises all errors by squaring them (thereby amplifying the effect of large deviations), Huber loss uses a hybrid approach that **balances sensitivity and robustness**.

Mathematically, the Huber loss behaves **quadratically** for small residuals (differences between observed and predicted values) and **linearly** for large residuals. This behaviour is governed by a threshold parameter known as **delta (δ)**:

- When the residual is less than δ in magnitude, the loss is computed as the square of the residual (just like in least squares).
- When the residual exceeds δ, the loss switches to a linear form, thus **limiting the influence** of large deviations.

This makes Huber loss especially useful in scenarios like radio astronomy data analysis, where **RFI spikes act as outliers**, but the underlying baseline trend still needs to be modelled accurately. By **down-weighting** these RFI peaks while still preserving the fit for the majority of the data, Huber loss helps achieve a **more reliable and robust baseline estimation** compared to standard least squares regression.

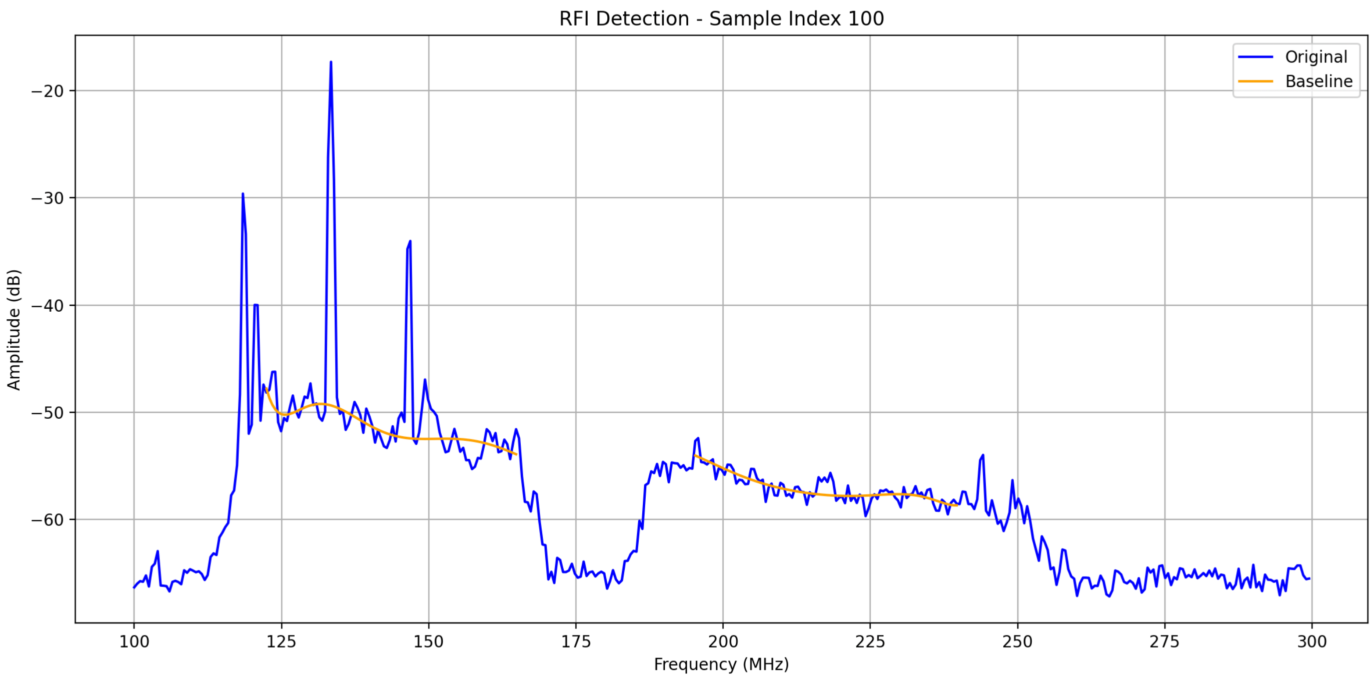

Figure 7.1 : baseline formation for band 2 based on LSR and Huber loss method

The plot above (figure 7.1) showcases the application of statistical baseline fitting techniques for detecting Radio Frequency Interference (RFI) in amplitude versus frequency data. The blue curve represents the **original observed signal** across a frequency range (in MHz), while the orange curve represents the **estimated baseline** computed using a robust fitting method.

To effectively identify RFI, it is essential to first model the underlying signal behaviour—i.e., the expected background amplitude trend in the absence of interference. This is done through **baseline fitting**, where a smooth curve is fit to the data using regression techniques.

Initially, a **Least Squares Regression (LSR)** approach is used. LSR attempts to find the curve that minimises the **sum of squared residuals**—that is, the squared difference between the actual amplitude values and the predicted baseline.

While LSR is efficient and accurate under ideal noise conditions, it is **highly sensitive to outliers**, which makes it **unsuitable** for datasets contaminated by strong RFI spikes. These outliers can skew the baseline upward, leading to inaccurate fits.

To overcome this limitation, the **Huber loss function** is employed. Huber loss combines the advantages of both LSR and absolute error loss. For residuals within a certain threshold (called **delta**), it behaves like LSR (squares the residuals). For larger residuals—often due to RFI peaks—it switches to linear behaviour, reducing their influence on the fit.

This **softens the impact of high-amplitude RFI spikes**, resulting in a more **robust and realistic baseline**. The baseline shown in orange is therefore generated using an **iterative Huber loss-based polynomial fitting**, which ensures that the baseline is unaffected by sharp RFI peaks, allowing those peaks to be accurately flagged as anomalies.

In this plot, one can observe multiple sharp spikes in the blue curve (especially around 120–150 MHz), which clearly deviate from the smooth orange baseline. These spikes are effectively ignored during baseline fitting due to the Huber loss mechanism, and they stand out as **potential RFI candidates**. The visual separation between the baseline and the peaks aids in automatic threshold-based RFI detection in subsequent steps.

Overall, the combination of LSR (for normal data) and Huber loss (for robust fitting in presence of RFI) provides a reliable and adaptive approach to detecting and isolating interference in radio observation data.

7.2 Python code :

```
import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

from datetime import datetime, timedelta

import os


# Defines frequency bands with their respective start and stop frequencies (in MHz)

band_frequencies = {

    "band 2": {"start": 100, "stop": 300},

    "band 3": {"start": 175, "stop": 575},

    "band 4": {"start": 500, "stop": 1000},

    "band 5": {"start": 800, "stop": 1700},

    "all bands": {"start": 0, "stop": 1500},

}


# Defines the frequency ranges used specifically for calculation within each band

calculation_band_frequencies = {

    "band 2": {"start": 122, "stop": 240},

    "band 3": {"start": 250, "stop": 500},

    "band 4": {"start": 550, "stop": 900},

    "band 5": {"start": 1000, "stop": 1450},

    "all bands": {"start": 110, "stop": 1450},

}


# Takes user input for the band and validates it

while True:
```

```
    user_band_input = input('Enter the band name (available bands: Band 2, Band 3, Band 4, Band 5, all bands): ').lower()

    if user_band_input in band_frequencies:

        selected_band_info = band_frequencies[user_band_input]

        start_frequency_in_MHz = selected_band_info["start"]

        stop_frequency_in_MHz = selected_band_info["stop"]

        print(f"Selected {user_band_input}: Start Frequency = {start_frequency_in_MHz} MHz, Stop Frequency = {stop_frequency_in_MHz} MHz")

        break

    else:

        print("Invalid band name. Please enter one of the bands mentioned here(Band 2, Band 3, Band 4, Band 5).")


# Converts start and stop frequencies from MHz to Hz and creates frequency bins

start_frequency = start_frequency_in_MHz * 1000000

stop_frequency = stop_frequency_in_MHz * 1000000

num_freq_bins = 401

freq_increment = (stop_frequency - start_frequency)/num_freq_bins

frequencies = np.array([start_frequency + i * freq_increment for i in range(num_freq_bins)])


# Takes user input for antenna and channel names

antenna_name = input("enter the antenna name:  ").upper()

channel_name = input("enter the channel name: ").upper()


# Takes user input for start and stop date, and validates the format and range

while True:

    try:

        start_date_str = input("Enter the start date (DD/MM/YYYY): ")
```

```
        stop_date_str = input("Enter the stop date (DD/MM/YYYY): ")
        user_start_date = datetime.strptime(start_date_str, "%d/%m/%Y").date()
        user_stop_date = datetime.strptime(stop_date_str, "%d/%m/%Y").date()
        if user_start_date > user_stop_date:
            print("Error: Start date cannot be after stop date. Please re-enter dates.")
        else:
            break
    except ValueError:
        print("Invalid date format. Please use DD/MM/YYYY.")

filename_array = []

# Filters relevant files based on filename pattern, frequency, and date range
for file_name in os.listdir('.'):
    if file_name.endswith('Hz.txt') and '_' in file_name:
        try:
            sections = file_name.split('_')
            if len(sections) < 5:
                print(f"  - Skipping malformed filename (not enough sections): {file_name}")
                continue
            file_day = sections[0].strip()
            file_month = sections[1].strip()
            file_year = sections[2].strip()
            file_date_str = f"{file_day}/{file_month}/{file_year}"
            file_date = datetime.strptime(file_date_str, "%d/%m/%Y").date()

            if user_start_date <= file_date <= user_stop_date:
```

```
            start_freq_section = sections[3].strip()
            stop_freq_section = sections[4].replace('Hz.txt', '').strip()
            file_start_freq = float(start_freq_section)



           file_stop_freq = float(stop_freq_section)


            if file_start_freq == start_frequency and file_stop_freq == stop_frequency:
                filename_array.append(file_name)
                print(f"  - Found relevant file: {file_name}")


        except (ValueError, IndexError) as e:
            print(f"  - Skipping file '{file_name}' due to parsing error: {e}")
            continue


filenames = filename_array


# Exits if no valid files were found
if not filenames:
    print("No files found matching the specified date and frequency range.")
    exit()
else:
    print(f"Successfully identified {len(filenames)} files for processing.")


# Loads data for a given file and filters by antenna and channel
def load_data(filename):
    try:
```

```
        df = pd.read_csv(filename, header=None, sep='\s+')
    except FileNotFoundError:
        print(f" file not found :{filename}")
        exit()
    filtered = df[(df[2] == antenna_name) & (df[3] == channel_name)].reset_index(drop=True)
    if filtered.empty:
        print(f'No data found for antenna {antenna_name} and channel {channel_name} in {filename}')
        return [], np.array([])
    time_strings = filtered[0].astype(str)+' '+filtered[1].astype(str)
    amplitude_matrix = filtered.iloc[:,13:].astype(float).values
    return time_strings.tolist(), amplitude_matrix


# Loads and stores all timestamps and amplitudes from matched files
all_time_strings = []
all_amplitudes = []


for filename in filenames:
    current_time_strings, current_amplitudes = load_data(filename)
    all_time_strings.extend(current_time_strings)
    if current_amplitudes.size > 0:
        all_amplitudes.append(current_amplitudes)


# Exits if no valid data was found
if not all_time_strings:
    print("No data loaded from any of the files. Exiting.")
    exit()
```

```
# Stacks all amplitude data into a single array
combined_amplitudes = np.vstack(all_amplitudes)

# Applies frequency range filtering based on calculation band
calc_band = calculation_band_frequencies[user_band_input]
calc_start_freq = calc_band["start"] * 1e6
calc_stop_freq = calc_band["stop"] * 1e6
calc_indices = np.where((frequencies >= calc_start_freq) & (frequencies <= calc_stop_freq))[0]

# Applies notch filter for band 2 by excluding frequencies from 165–195 MHz
if user_band_input == "band 2":
    notch_start = 165e6
    notch_stop = 195e6
    calc_indices = calc_indices[
        ~((frequencies[calc_indices] >= notch_start) & (frequencies[calc_indices] <= notch_stop))
    ]

# Extracts filtered frequencies and amplitudes
filtered_frequencies = frequencies[calc_indices]
filtered_amplitudes = combined_amplitudes[:, calc_indices]

# Converts amplitude from dB to linear milliwatts
amplitudes_linear_mW = 10 ** (combined_amplitudes / 10)

# Parameters for baseline fitting and RFI detection
poly_order = 10
```

```
threshold_sigma = 3

delta = 1.0

max_iter = 15


# Applies robust polynomial fitting with Huber loss to a single row

def process_row(y, x, delta):

    coeffs = robust_polyfit(x, y, poly_order, max_iter=max_iter, delta=delta)

    y_fit = np.polyval(coeffs, x)

    residual = y - y_fit

    std = np.std(residual)

    rfi_mask = np.abs(residual) > threshold_sigma * std

    return y_fit, rfi_mask


# Computes Huber weights for residuals

def huber_weights(residuals, delta):

    abs_residuals = np.abs(residuals)

    weights = np.ones_like(residuals)

    weights[abs_residuals > delta] = delta / abs_residuals[abs_residuals > delta]

    return weights


# Performs robust polynomial fitting using iterative reweighting

def robust_polyfit(x, y, degree, max_iter, delta):

    weights = np.ones_like(y)

    for _ in range(max_iter):

        coeffs = np.polyfit(x, y, degree, w=weights)

        y_fit = np.polyval(coeffs, x)

        residuals = y - y_fit
```

```
        weights = huber_weights(residuals, delta)
    return coeffs


# Processes each file and performs baseline fitting and RFI detection
for filename in filenames:
    print(f"Processing file: {filename}")
    time_strings, amplitudes = load_data(filename)
    if len(amplitudes) == 0:
        continue
    baselines = []
    rfi_masks = []

    for y in amplitudes:
        x = np.arange(len(y))
        y_fit, rfi_mask = process_row(y, x, delta)
        baselines.append(y_fit)
        rfi_masks.append(rfi_mask)

    baselines = np.array(baselines)
    rfi_masks = np.array(rfi_masks)

    sample_index = int(input(f"Select time index (0 to {len(amplitudes)-1}): "))
    y_full = amplitudes[sample_index]
    y_calc = y_full[calc_indices]
    x_calc = np.arange(len(y_calc))

    y_fit_calc, rfi_mask = process_row(y_calc, x_calc, delta)
```

```python
    y_fit_full = np.full_like(y_full, np.nan)
    y_fit_full[calc_indices] = y_fit_calc

    file_frequencies = frequencies
    plt.figure(figsize=(12, 6))
    plt.plot(frequencies / 1e6, y_full, label="Original", color='blue')
    plt.plot(frequencies / 1e6, y_fit_full, label="Baseline", color='orange')

    # Optionally plot RFI points (commented out)
    # plt.scatter(frequencies[calc_indices][rfi_mask] / 1e6, y_calc[rfi_mask], color='red', label='RFI Detected', s=25)

    plt.title(f"RFI Detection - Sample Index {sample_index}")
    plt.xlabel("Frequency (MHz)")
    plt.ylabel("Amplitude (dB)")
    plt.legend()
    plt.grid(True)
    plt.tight_layout()
    plt.show()
```

8. FREQUENCY VS TIME OCCUPANCY PER FREQUENCY BIN

8.1 What are the stats given by the frequency vs time occupancy plots ?

The Frequency vs. Time Occupancy plot provides a **statistical overview of RFI activity across different frequency bins**. It shows the percentage of time each frequency bin was occupied by an RFI event over a given observation period. The x-axis represents the frequency in MHz, while the y-axis displays the **occupancy percentage**, which quantifies **how frequently a particular frequency bin exceeded the RFI threshold** (typically defined using a baseline plus a multiple of standard deviation, such as 3σ).

In the analysis of Radio Frequency Interference (RFI), **frequency vs time occupancy plots** provide a statistical view of how often each frequency bin is contaminated with interference. In simpler terms, the **occupancy percentage** reflects how persistently a frequency is "active" with signals stronger than the normal background (baseline) level, specifically **above 3σ** (3 standard deviations from baseline noise).

This plot is generated by scanning across all time-stamped measurements and counting the number of times the amplitude (the residual of the amplitude) at each frequency bin exceeded the detection threshold. The ratio of this count to the total number of time instances is then expressed as a **percentage**, indicating how often each frequency bin was affected by interference.

This visualisation is especially useful for **identifying persistently contaminated frequencies** and **understanding RFI distribution** across the spectrum. Frequencies with high occupancy percentages may correspond to known transmitters, satellite bands, or persistent noise sources. Meanwhile, frequency regions with near-zero occupancy can be considered cleaner and more reliable for scientific analysis.

By presenting this data in a concise bar-graph format, the plot enables **comparative assessment of spectral cleanliness**, guiding astronomers or engineers to focus on less contaminated bands or to plan exclusion zones for RFI mitigation techniques.

8.2 Frequency vs time occupancy per spectral channel for band - 2:

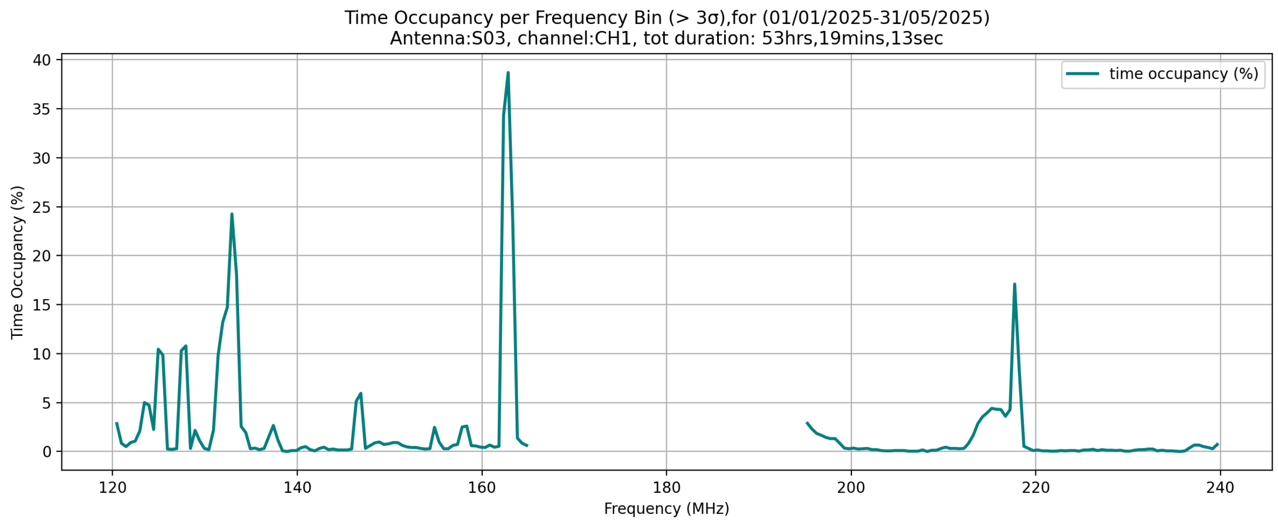

Figure 8.1: frequency vs time occupancy plot for band 2 showing the notch with the discontinuity

The above plot illustrates the **time occupancy percentage per frequency bin** for **Band 2**, calculated using data from **Antenna S03 and Channel CH1** over the user-specified duration: **01/01/2025 to 31/05/2025**. The **total observation time** covered in this plot is approximately **53 hours, 19 minutes, and 13 seconds**.

The **x-axis** represents the frequency in **MHz**, while the **y-axis** shows the **percentage of time** that each frequency bin recorded an RFI event—defined here as an amplitude exceeding the **3-sigma threshold** above the fitted baseline. This threshold ensures only statistically significant deviations are considered as interference.

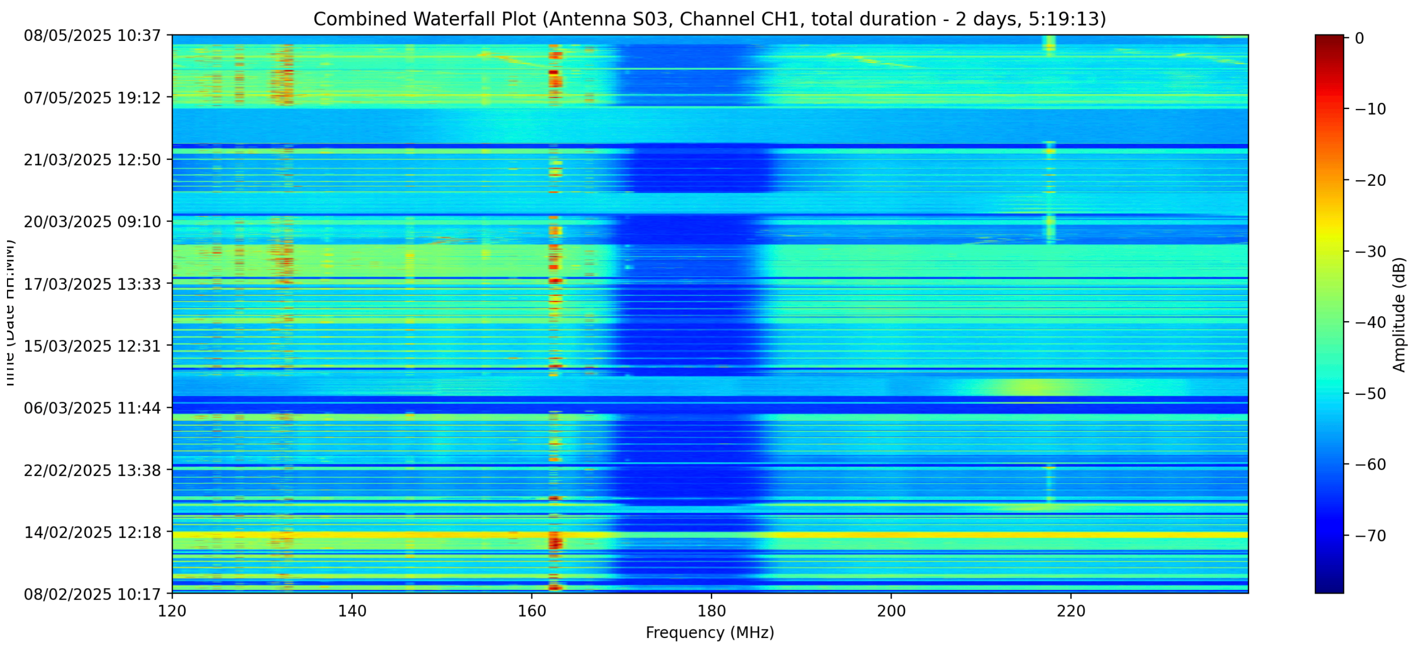

Figure 8.2: Gray scale plot for band 2 (same antenna , channel and date specifications as the time occupancy

The gray scale plot or the water fall plot has been attached for comparison with the time occupancy (per spectral channel) plot of the band 2 .

Also band 2 , has a peculiar band shape in the frequency range 165 MHz to 185 MHz , a typical notch and that frequency range has been ignored and hence we see the discontinuity in the time occupancy curve .

Overall, this plot highlights the **non-uniform distribution** of RFI across Band 2 and emphasises the importance of **bin-wise statistical monitoring** in radio frequency analysis.

8.3 Frequency vs time occupancy per spectral channel for band - 3:

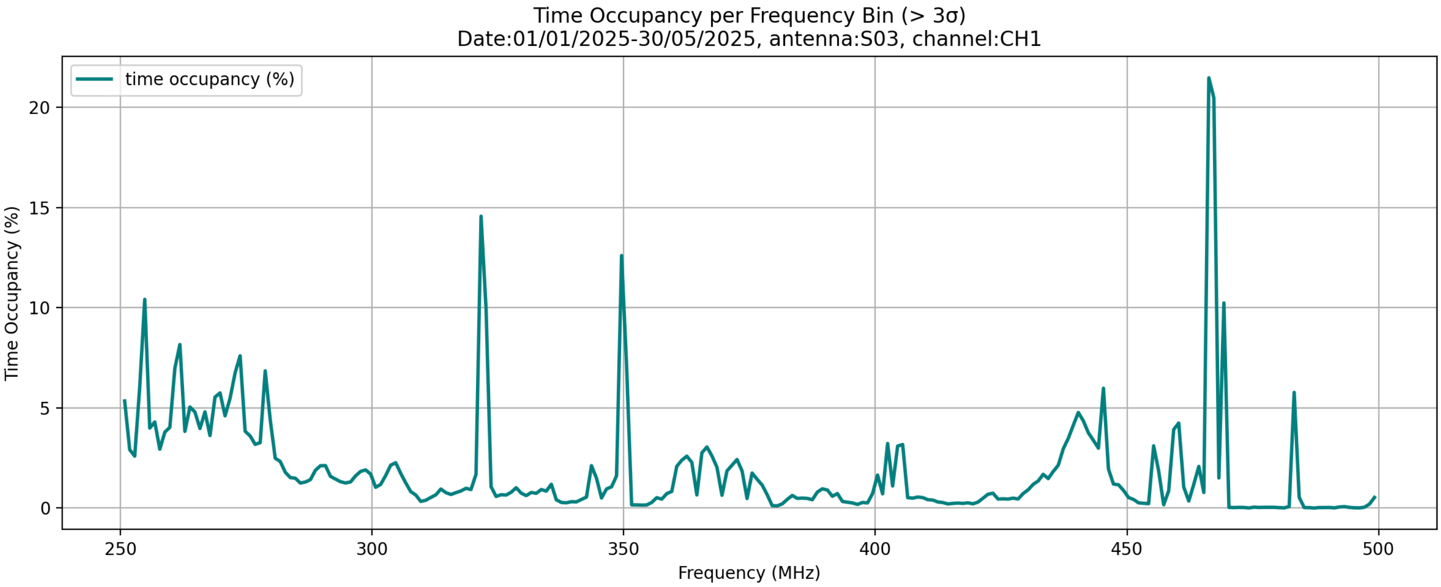

Figure 8.3: frequency vs time occupancy plot for band 3 for a duration of 5 months

The above plot shows the **time occupancy percentage per frequency bin** for **Band 3**, based on RFI detections from **Antenna S03 and Channel CH1**, spanning the observation period **01/01/2025 to 30/05/2025**. The x-axis represents the **frequency in MHz**, while the y-axis shows the **percentage of time** each frequency bin was contaminated by RFI (i.e., the signal exceeded **3σ** above the baseline).

This occupancy metric helps quantify **how often** each spectral bin is affected by interference.

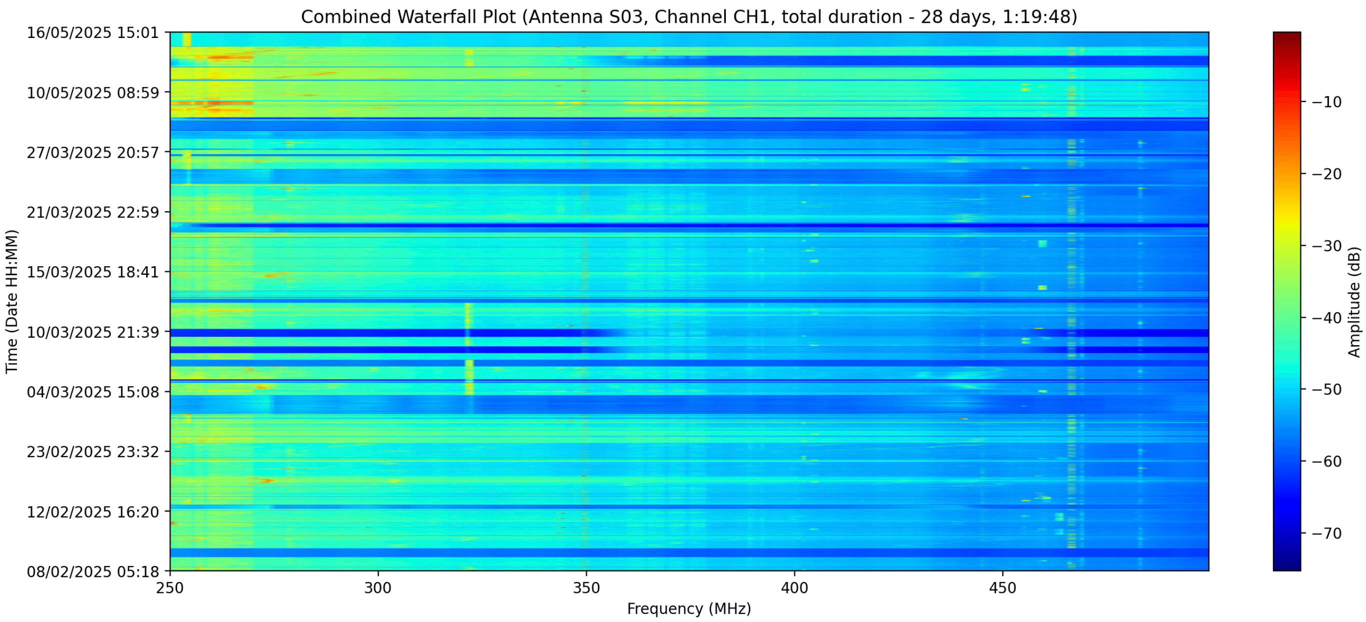

Figure 8.4: Gray scale plot for band 3 , (same antenna , channel and date specifications as the time occupancy plot)

The gray scale plot or the water fall plot has been attached for comparison with the time occupancy (per spectral channel) plot of the band 3 .

In summary, Band 3 shows a **non-uniform distribution of RFI**, with clear interference signatures at certain frequencies that merit attention in future observation planning.

8.4 Frequency vs time occupancy per spectral channel for band - 4:

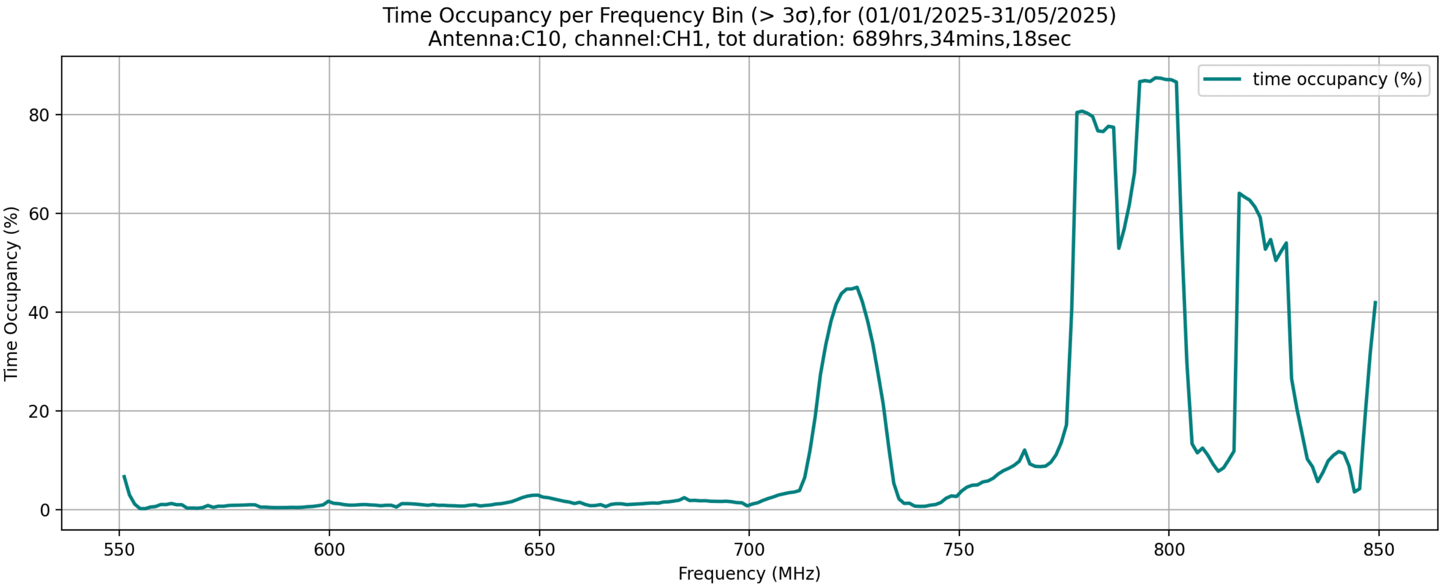

Figure 8.5 frequency vs time occupancy (per spectral Channel) plot for band 4

The plot shown above , is the band 4 plot of a duration of five months , dated : 01/01/2025 - 31/05/2025. The **X-axis** is frequency (MHz), and the **Y-axis** is the **% of time** that the signal in that bin was above the 3σ threshold.

The methodology used for band 2, 3 and 5 worked fine with band 4's 2024 data as in band 4, the known persistent RFI's frequency ranges 772 MHz - 803 MHz and 842 MHz to 900 MHz were ignored in the baseline formation just similar to the notch in band 2 (165 MHz - 185MHz) .

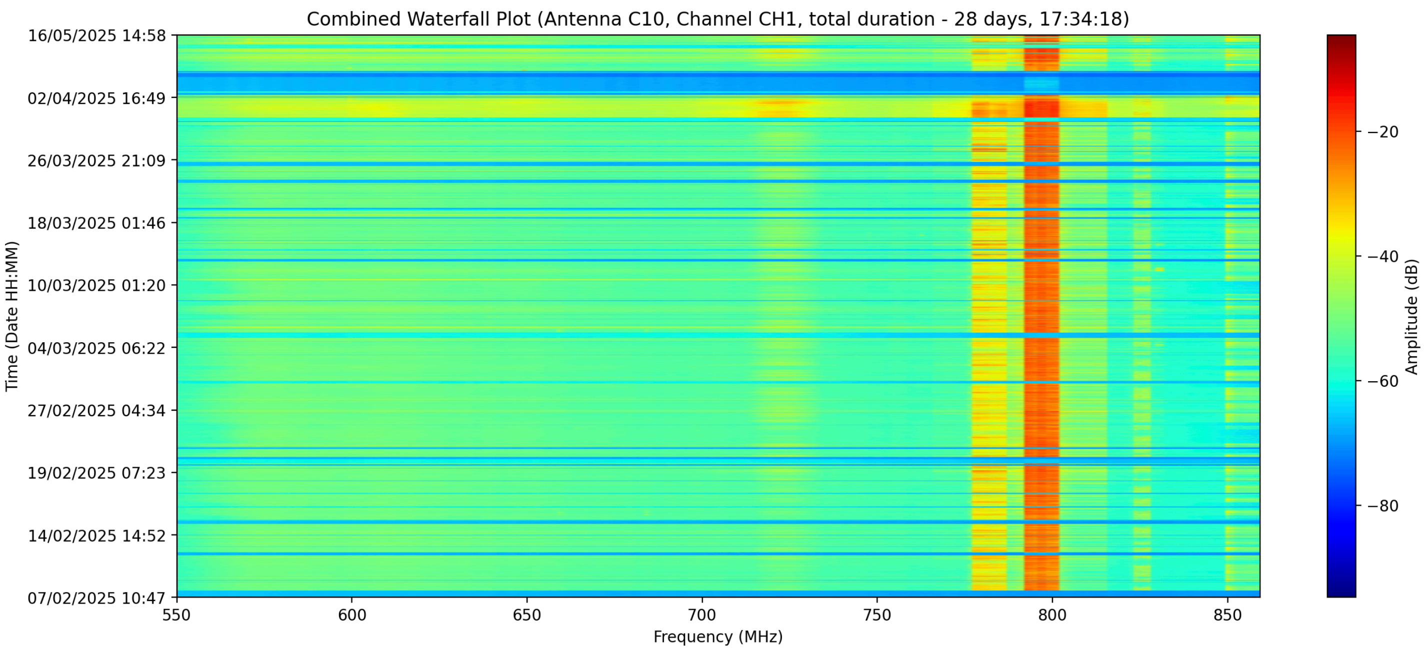

Figure 8.6 Gray scale plot for the band 4 , for a duration of 5 months dated - 01/01/2025 - 31/05/2025

But the same methodology couldn’t be applied to the band 4’s data of 2025, because of the fact that frequency range between the mentioned data was affected by RFI as well. This significantly shoot up the three sigma value this affecting the baseline , residual and hence the plot .

The solution was to **borrow the 3σ threshold from older clean data (2016)**. That dataset had a very clean band shape due to the fact that the population back then was less and the electronic gadgets usage was not as much as the present day’s applications thus contributing to very less or almost no RFI in most of the regions of the band .

Applying the **2016-derived 3σ threshold** to the 2025 data brought the detection back on track, allowing **every significant RFI spike to be correctly captured**, as seen in the plot where time occupancy reaches over 95**%** for some bins.

The **near-zero values elsewhere** indicate the rest of the band was relatively clean ,once a reliable threshold was used.

The waterfall plot confirms the **exact same frequency zones** were consistently active across time.

It provides a **visual timeline** of RFI events that matches the **statistical detection** from the time occupancy plot.

8.5 Python Code :

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from matplotlib import ticker
from datetime import datetime, timedelta
import os

# Defines the frequency range (in MHz) for each band
band_frequencies = {
    "band 2": {"start": 100, "stop": 300},
    "band 3": {"start": 175, "stop": 575},
    "band 4": {"start": 500, "stop": 1000},
    "band 5": {"start": 800, "stop": 1700},
    "all bands": {"start": 0, "stop": 1500},
}

# Defines the narrower frequency range used for calculations for each band
calculation_band_frequencies = {
    "band 2": {"start": 120, "stop": 240},
    "band 3": {"start": 250, "stop": 500},
    "band 4": {"start": 550, "stop": 850},
    "band 5": {"start": 1000, "stop": 1450},
    "all bands": {"start": 120, "stop": 1450},
}

# Prompts user to input a valid band name and retrieves corresponding frequency range
while True:
    user_band_input = input('Enter the band name (available bands: Band 2, Band 3, Band 4, Band 5,
all bands): ').lower()

    if user_band_input in band_frequencies:
```

```
        selected_band_info = band_frequencies[user_band_input]
        start_frequency_in_MHz = selected_band_info["start"]
        stop_frequency_in_MHz = selected_band_info["stop"]
        print(f"Selected {user_band_input}: Start Frequency = {start_frequency_in_MHz} MHz, Stop Frequency = {stop_frequency_in_MHz} MHz")
        break
    else:
        print("Invalid band name. Please enter one of the bands mentioned here(Band 2, Band 3, Band 4, Band 5).")

# Converts start/stop frequency from MHz to Hz and generates frequency bins
start_frequency = start_frequency_in_MHz * 1_000_000
stop_frequency = stop_frequency_in_MHz * 1_000_000
num_freq_bins = 401
freq_increment = (stop_frequency - start_frequency)/num_freq_bins
frequencies = np.array([start_frequency + i * freq_increment for i in range(num_freq_bins)])

# Gets user input for antenna and channel names
antenna_name =input("Enter the antenna name : ").upper()
channel_name = input("Enter the channel name : ").upper()

# Prompts user to enter a valid start and stop date for filtering
while True:
    try:
        start_date_str = input("Enter the start date (DD/MM/YYYY): ")
        stop_date_str = input("Enter the stop date (DD/MM/YYYY): ")

        user_start_date = datetime.strptime(start_date_str, "%d/%m/%Y").date()
        user_stop_date = datetime.strptime(stop_date_str, "%d/%m/%Y").date()

        if user_start_date > user_stop_date:
            print("Error: Start date cannot be after stop date. Please re-enter dates.")
        else:
```

```
            break
    except ValueError:
        print("Invalid date format. Please use DD/MM/YYYY.")

# Initializes list to store filenames matching date and frequency range
filename_array = []

# Loops through all files in current directory to find relevant files
for file_name in os.listdir('.'):
    if file_name.endswith('Hz.txt') and '_' in file_name:
        try:
            sections = file_name.split('_')
            if len(sections) < 5:
                print(f"  - Skipping malformed filename (not enough sections): {file_name}")
                continue
            file_day = sections[0].strip()
            file_month = sections[1].strip()
            file_year = sections[2].strip()
            file_date_str = f"{file_day}/{file_month}/{file_year}"
            file_date = datetime.strptime(file_date_str, "%d/%m/%Y").date()

            if user_start_date <= file_date <= user_stop_date:
                start_freq_section = sections[3].strip()
                stop_freq_section = sections[4].replace('Hz.txt', '').strip()
                file_start_freq = float(start_freq_section)
                file_stop_freq = float(stop_freq_section)

                if file_start_freq == start_frequency and file_stop_freq == stop_frequency:
                    filename_array.append(file_name)
                    print(f"  - Found relevant file: {file_name}")

        except (ValueError, IndexError) as e:
            print(f"  - Skipping file '{file_name}' due to parsing error: {e}")
```

```
            continue

# Assigns filtered filenames to a variable
filenames = filename_array

# Checks if any matching files are found
if not filenames:
    print("No files found matching the specified date and frequency range.")
    exit()
else:
    print(f"Successfully identified {len(filenames)} files for processing.")

# Defines a function to load amplitude and time data for a given file
def load_data(filename):
    try:
        df = pd.read_csv(filename, header=None, sep=r'\s+')
    except FileNotFoundError:
        print(f"File not found: {filename}")
        return None, None

    # Filters rows based on selected antenna and channel
    filtered = df[(df[2] == antenna_name) & (df[3] == channel_name)].reset_index(drop=True)
    if filtered.empty:
        print(f"Skipping {filename}: no data for antenna {antenna_name} and channel
{channel_name}")
        return None, None

    # Extracts amplitude matrix and time strings
    amplitude_matrix = filtered.iloc[:, 13:].astype(float).values
    time_strings = filtered[0].astype(str) + ' ' + filtered[1].astype(str)

    return time_strings.tolist(), amplitude_matrix
```

```
# Initializes lists to hold all time and amplitude data
all_time_strings = []
all_amplitudes = []

# Loads data from each relevant file and aggregates it
for filename in filenames:
    current_time_strings, current_amplitudes = load_data(filename)
    if current_time_strings is None or current_amplitudes is None:
        continue  # skip this file

    all_time_strings.extend(current_time_strings)
    if current_amplitudes.size > 0:
        all_amplitudes.append(current_amplitudes)

# Exits if no valid data is found
if not all_time_strings:
    print("No data loaded from any of the files. Exiting.")
    exit()

# Combines amplitude matrices from all files into one
combined_amplitudes = np.vstack(all_amplitudes)

# Extracts calculation range and indexes based on selected band
calc_band = calculation_band_frequencies[user_band_input]
calc_start_freq = calc_band["start"] * 1e6
calc_stop_freq = calc_band["stop"] * 1e6
calc_indices = np.where((frequencies >= calc_start_freq) & (frequencies <= calc_stop_freq))[0]

# Filters frequency and amplitude data based on calculation band
filtered_frequencies = frequencies[calc_indices]
filtered_amplitudes = combined_amplitudes[:, calc_indices]

# Sets threshold and Huber parameters
```

```
threshold_sigma = 3
poly_order = 10
delta = 1.0
max_iter = 15

# Defines function to process a single row of data using robust fitting
def process_row(y, x, delta):
    coeffs = robust_polyfit(x, y, poly_order, max_iter=max_iter, delta=delta)
    y_fit = np.polyval(coeffs, x)
    residual = y - y_fit
    if user_band_input == "band 4":
        std = 0.82666667
    else:
        std = np.std(residual)
    rfi_mask = np.abs(residual) > threshold_sigma * std
    return y_fit, rfi_mask

# Defines function to compute Huber weights based on residuals and delta
def huber_weights(residuals, delta):
    abs_residuals = np.abs(residuals)
    weights = np.ones_like(residuals)
    weights[abs_residuals > delta] = delta / abs_residuals[abs_residuals > delta]
    return weights

# Defines function to perform iterative robust polynomial fitting using Huber loss
def robust_polyfit(x, y, degree, max_iter, delta):
    weights = np.ones_like(y)
    for _ in range(max_iter):
        coeffs = np.polyfit(x, y, degree, w=weights)
        y_fit = np.polyval(coeffs, x)
        residuals = y - y_fit
        weights = huber_weights(residuals, delta)
    return coeffs
```

```
# Creates RFI mask for each row in amplitude matrix
x = np.arange(filtered_amplitudes.shape[1])
masks_all = []
for row in filtered_amplitudes:
    _, mask_row = process_row(row, x, delta)
    masks_all.append(mask_row)

# Stacks masks to form final RFI mask matrix
mask = np.vstack(masks_all)
mask_flat = mask.T.flatten()

# Converts string timestamps to datetime and extracts only time
time_objects = [datetime.strptime(t, "%d/%m/%Y %H:%M:%S") for t in all_time_strings]
time_objects_only = [t.time() for t in time_objects]

# Converts time to numeric (minutes) format
def time_to_minutes(t):
    return t.hour * 60 + t.minute + t.second / 60
times_numeric = [time_to_minutes(t) for t in time_objects_only]
times_numeric_repeated = np.tile(times_numeric, len(calc_indices))

# Calculates RFI occupancy percentage per time step
rfi_bandwidth_percentages_per_time = np.sum(mask, axis=1) / mask.shape[1] * 100

# Calculates total and threshold-exceeding values for percentage
total_values = filtered_amplitudes.size
above_threshold_values = np.count_nonzero(mask)
percentage_above_threshold = (above_threshold_values / total_values) * 100

# Calculates occupancy percentage per frequency bin
num_time_steps = filtered_amplitudes.shape[0]
time_occupancy_percentages = np.sum(mask, axis=0) / num_time_steps * 100
```

```
# Prints frequency vs time occupancy percentages(commented out)
#print("\nTime Occupancy (%) per Frequency Bin (where residual > 3σ):")
#for freq, occupancy in zip(filtered_frequencies, time_occupancy_percentages):
 #   if occupancy > 0:
  #       print(f"  Frequency {freq/1e6:.2f} MHz: {occupancy:.2f}% of time")

# Calculates overall RFI presence per time step
any_rfi_per_time = np.any(mask, axis=1)
num_time_steps_with_rfi = np.count_nonzero(any_rfi_per_time)
total_time_steps = filtered_amplitudes.shape[0]
total_rfi_time_percentage = (num_time_steps_with_rfi / total_time_steps) * 100

# Displays total percentage of time affected by RFI
print(f"\nTotal percentage of time with any frequency bin above 3σ: {total_rfi_time_percentage:.2f}
%")

# Prepares data for scatter plotting
freqs_repeated = np.repeat(filtered_frequencies, filtered_amplitudes.shape[0])
freqs_plot = freqs_repeated[mask_flat] / 1e6
times_plot = np.array(times_numeric_repeated)[mask_flat]

# Extracts date string for use in plot title
if filenames:
    date_parts = filenames[0].split('_')[:3]
    file_date_cleaned = '_'.join(part.strip() for part in date_parts)
    file_date_obj = datetime.strptime(file_date_cleaned, "%d_%m_%Y").date()
    date_str_for_title = file_date_obj.strftime("%B %d, %Y")
else:
    date_str_for_title = "Unknown Date"

# Defines function to load timestamps from all rows of a file
def load_timestamps_all_rows(filename):
```

```
    try:
        df = pd.read_csv(filename, header=None, sep=r'\s+')
    except FileNotFoundError:
        print(f"File not found: {filename}")
        return []

    time_strings = df[0].astype(str) + ' ' + df[1].astype(str)
    try:
        return [datetime.strptime(t, "%d/%m/%Y %H:%M:%S") for t in time_strings]
    except Exception as e:
        print(f"Failed to parse timestamps in {filename}: {e}")
        return []

# Calculates total observation duration with segmentation check
total_duration = timedelta(0)
for file in filenames:
    timestamps = load_timestamps_all_rows(file)
    if len(timestamps) < 2:
        print(f"{file}: Not enough timestamps for duration calculation.")
        continue

    timestamps.sort()
    file_duration = timedelta(0)
    segment_start = timestamps[0]
    for i in range(1, len(timestamps)):
        delta = timestamps[i] - timestamps[i - 1]
        if delta > timedelta(seconds=2):
            file_duration += timestamps[i - 1] - segment_start
            segment_start = timestamps[i]
    file_duration += timestamps[-1] - segment_start
    print(f"{file}: duration = {file_duration}")
    total_duration += file_duration
```

```
# Converts total duration into hours, minutes, and seconds
hours = total_duration.total_seconds() // 3600
minutes = (total_duration.total_seconds() % 3600) // 60
seconds = total_duration.total_seconds() % 60

# Prints total observation time
print(f"\nTotal observation duration across all files (regardless of antenna/channel, with >2s
segmentation): {int(hours)} hours, {int(minutes)} minutes, {int(seconds)} seconds.")

# Plots time occupancy percentage per frequency bin
plt.figure(figsize=(12, 5))
plt.plot(filtered_frequencies / 1e6, time_occupancy_percentages, color='teal', linewidth=2,
label='time occupancy (%)')
plt.xlabel("Frequency (MHz)")
plt.ylabel("Time Occupancy (%)")
plt.title(f"Time Occupancy per Frequency Bin (> 3σ),for ({start_date_str}-{stop_date_str})
\nAntenna:{antenna_name}, channel:{channel_name}, tot duration: {int(hours)}hrs,{int(minutes)}
mins,{int(seconds)}sec")
plt.grid(True)
plt.tight_layout()
plt.legend()
plt.show()
```

9. THREE SIGMA COMPARISON OVER THE YEARS

9.1 Comparison of 3 sigma over the years

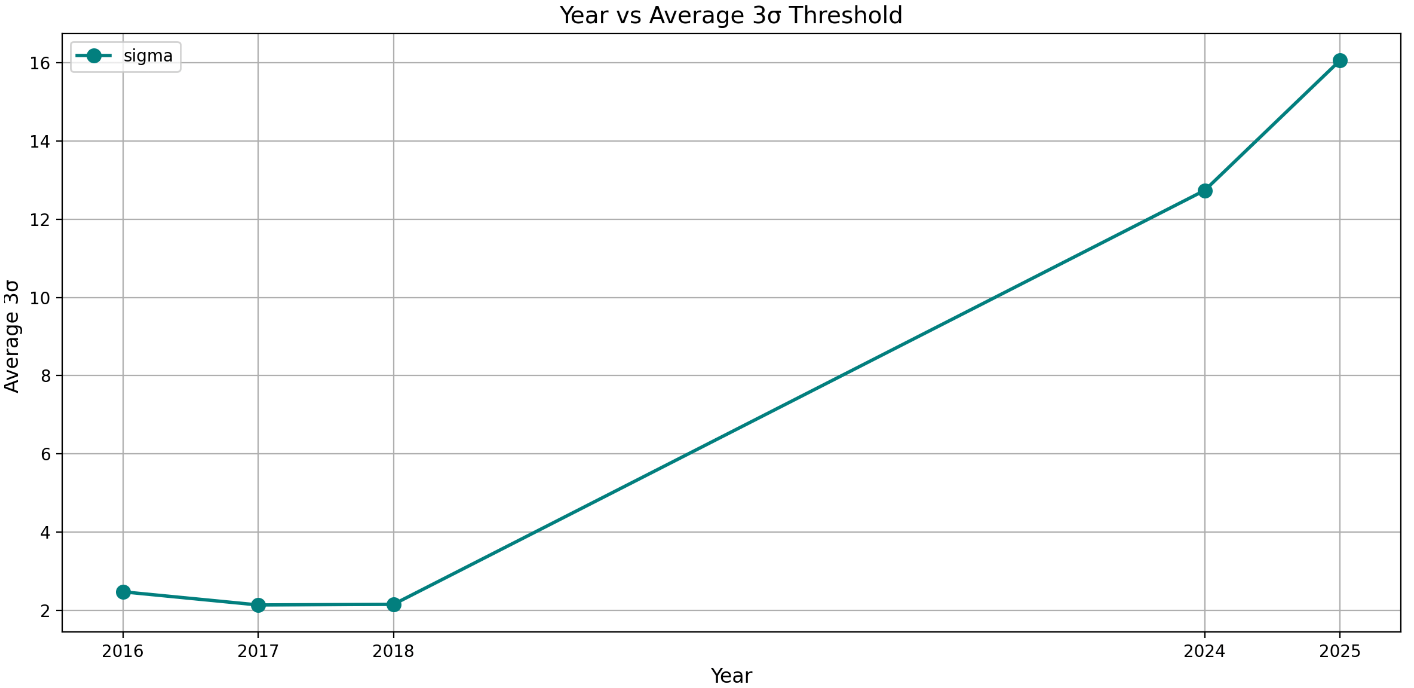

Figure: 9.1: Year vs average 3 sigma plot showing the variation of 3 sigma over the years

The **plot** shows the variation of average **3σ (three-sigma) threshold values** across the years from 2016 to 2025 for **band 4**. In the early years—2016 through 2018—the sigma values are relatively low and consistent, staying close to 2.5. This indicates that the spectral data during those years was cleaner, with lower variance in the baseline, and thus a stable and reliable threshold for RFI detection.

However, a sharp increase in sigma is observed starting from 2019, and more drastically in 2024 and 2025, where the 3σ values rise dramatically reaching a peak in 2025. This rise reflects increased contamination of the band with persistent RFI in recent years, which artificially inflates the baseline noise level. As sigma represents the standard deviation of the fitted baseline, a corrupted baseline leads to a significantly higher sigma, which can weaken the effectiveness of the 3σ threshold as a discriminator for RFI.

These 3 sigma values are taken on the basis of baseline formation using the Huber loss technique , and the standard deviation of the residuals is taken across all the spectral channels (that is the entire band width)

Thus, the plot clearly illustrates how **the reliability of using a data-driven sigma threshold diminishes over time**, especially in heavily polluted bands, necessitating alternative approaches such as referencing cleaner bands or older reference data.

9.2 Comparison of 3 sigma over the years after implying the cleaner band approach for 2024 and 2025.

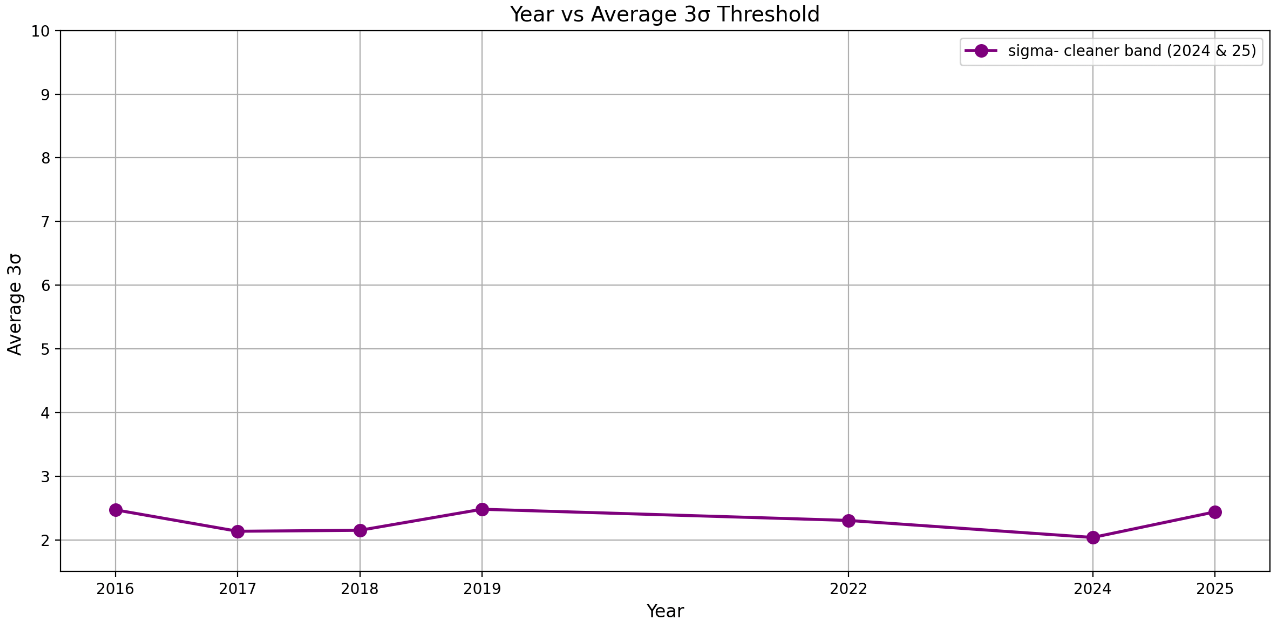

Figure 9.2: year vs avg 3 sigma plot showing the 3 sigma of the cleaner band for 2024 and 2025

The slide presents a comparative analysis of the **3σ (three-sigma) threshold values** across several years—2016, 2017, 2018, 2019, 2022, 2024, and 2025—for **Band 4** radio data. This threshold is critical in RFI detection, as it marks the cutoff above which signal amplitudes are considered abnormal or potentially contaminated by interference.

In this analysis, a consistent **"cleaner band" approach** was implemented for the years 2024 and 2025, where only the uncorrupted, relatively RFI-free parts of the spectrum were used to calculate the 3σ values. This was necessary because some parts of the 2025 dataset were corrupted particularly in the regions between persistent RFI zones leading to an overestimation of the noise floor and hence an ineffective threshold for detecting actual RFI. The cleaner segments were therefore isolated manually or algorithmically to ensure accurate threshold estimation.

The plotted trend shows that the average 3σ values remain **remarkably consistent across years** when cleaner data is used, with minor variations. This consistency validates the cleaner band approach as a **reliable fallback** strategy when no historical reference dataset is available.

Essentially, if an older clean dataset (like 2016) cannot be reused, applying this cleaner-band extraction method allows one to **reliably derive detection thresholds** even from partially corrupted data, as was done for 2024 and 2025 .

Thus, this graph not only highlights the **stability of sigma values** over time under careful selection but also emphasises the **robustness of the cleaner band method** in maintaining threshold accuracy despite varying data quality.

10. TIME OCCUPANCY AND BANDWIDTH OCCUPANCY

```
Bandwidth Occupancy (%) per Time Step (i.e., how much of the band had RFI):
  Time 10:17:58 → 2.92% of band had RFI
  Time 10:18:58 → 2.50% of band had RFI
  Time 10:19:58 → 1.67% of band had RFI
  Time 10:20:58 → 8.33% of band had RFI
  Time 10:21:58 → 12.08% of band had RFI
  Time 10:22:58 → 7.08% of band had RFI
  Time 10:23:58 → 15.00% of band had RFI
  Time 10:24:58 → 8.33% of band had RFI
  Time 10:25:58 → 10.42% of band had RFI
  Time 10:26:58 → 9.58% of band had RFI
  Time 10:27:58 → 1.67% of band had RFI
  Time 10:28:58 → 0.83% of band had RFI
  Time 10:29:58 → 3.75% of band had RFI
  Time 10:30:58 → 3.33% of band had RFI
  Time 10:31:58 → 2.50% of band had RFI
  Time 10:32:58 → 2.08% of band had RFI
```

Figure 10.1: shows the bandwidth occupancy for band 2, for a single file dated : 19/04/2024

The graph presents the variation of bandwidth occupancy percentage over time for Band-2 on April 19, 2024. It captures how much of the observed bandwidth was affected by radio frequency interference (RFI) at each recorded time point. The y-axis represents the percentage of bandwidth flagged as RFI, while the x-axis corresponds to specific timestamps throughout the day.

The data shows noticeable fluctuations in occupancy, with several spikes crossing 40%, indicating periods of significant RFI activity. This time-resolved visualisation enables quick identification of interference-heavy periods and helps in understanding the temporal distribution and intensity of RFI within the selected frequency band.

```
Time Occupancy (%) per Frequency Bin (where residual > 3σ):
  Frequency 120.45 MHz: 9.21% of time
  Frequency 120.95 MHz: 5.70% of time
  Frequency 121.45 MHz: 3.95% of time
  Frequency 121.95 MHz: 4.39% of time
  Frequency 122.44 MHz: 4.82% of time
  Frequency 122.94 MHz: 6.58% of time
  Frequency 123.44 MHz: 7.89% of time
  Frequency 123.94 MHz: 9.21% of time
  Frequency 124.44 MHz: 7.02% of time
  Frequency 124.94 MHz: 13.60% of time
  Frequency 125.44 MHz: 13.60% of time
  Frequency 125.94 MHz: 1.75% of time
  Frequency 126.43 MHz: 1.75% of time
  Frequency 126.93 MHz: 3.07% of time
```

Figure 10.2 : shows the time occupancy for each spectral channel of band 2 , dated : 19/04/2024

The statistics display the percentage of time that specific frequency bins experienced radio frequency interference (RFI), measured by how often the residual signal exceeded a 3σ threshold. These values give insight into how persistently certain frequencies are impacted by RFI over the observation period.

The plot illustrates the time occupancy percentage of radio frequency interference (RFI) across different frequency bins. Each bar represents how frequently a particular frequency experienced interference over the observation period, calculated by detecting instances where the signal exceeded a 3-sigma threshold above the baseline. Higher occupancy percentages indicate frequencies that are more consistently impacted by RFI, while lower values correspond to cleaner regions of the spectrum. This analysis provides a clear statistical overview of which parts of the frequency spectrum are most affected by interference, offering valuable insight for identifying persistently contaminated regions and optimising future data collection or mitigation strategies.

11. FUTURE SCOPES OF MY PROJECT

The project primarily revolves around statistical evaluation of RFI patterns using historical spectral data. In future extensions, the methodology can be scaled to deliver a **comprehensive descriptive analysis** across various **spectral channels**, **frequency bands**, and **time stamps**, providing deeper insights into long-term RFI behaviour.

The robust detection techniques—particularly the use of **Huber loss-based baseline estimation** and **least squares regression (LSR)**—can be **extended to LTA (Long Term Accumulation) files**, broadening the applicability of this framework to larger, long-duration datasets.

Additionally, the pipeline can be adapted to automatically **generate RFI logs for all GTAC (GMRT Time Allocation Committee) observations**, ensuring consistent monitoring and documentation of interference events across observing sessions.

12. Websites and research papers used for reference:

1. http://www.ncra.tifr.res.in/ncra/gmrt/about-gmrt/introducing-gmrt-1/introducing-gmrt
2. http://www.ncra.tifr.res.in/ncra/gmrt
3. https://www.mdpi.com/2226-4310/8/2/51
4. https://agupubs.onlinelibrary.wiley.com/doi/full/10.1029/2004RS003172
5. https://www.mdpi.com/1424-8220/19/2/306?utm_source=chatgpt.com
6. https://www.cantorsparadise.com/huber-loss-why-is-it-like-how-it-is-dcbe47936473
7. https://www.investopedia.com/terms/l/least-squares-method.asp