The Sun's Atmosphere: Microwave data analysis project

Taotao Yang
School of Physics & Astronomy, University of Glasgow
(Dated: April 02, 2023; Edit: April 11, 2023)

The project focus on the data analysis of the source data obtained from Nobeyama Radio Polarimeters (NoRP) and Radio Solar Telescope Network (RSTN). The project use python to read, organize, process, fit, and plot the radio data. The investigated data is centered around a flare event on October 28th, 2013, peaking at 01:59:38 UTC, with a frequency range of 0.245 GHz to 80 GHz. The project produce curve fits on Gyrosynchrotron emissions by adjusting the source data with a fitted plasma emission model. The Gyrosynchrotron fit after plasma model correction shows a better curve fit result than the original fit, where the plasma emissions at lower frequencies are included.

I. Introduction

The sun is the major driving force of space weather, and the study of flares and their associated activities could provide us with important insights on the generation mechanics, the profiles, and influences of these electromagnetic emissions coming from the solar atmosphere [1]. The effect of solar activities on Earth includes the disruptions to communications and energy infrastructures, increased radiation to objects in near earth space, and random incursions within circuits and logic processors, which would produces undesired logic outputs and readings due to single event offsets [2]. And given the distance between the Earth and the Sun, efforts has been made to use machine learning to analyse the images of solar atmosphere, from which solar flares can be predicted [3].

This data analysis project will investigate the role of plasma and gyrosynchrtron emissions in the context of microwave bands. The project will begin with theoretical backgrounds on plasma emissions and gyrosynchrtron emissions, in terms of their mathematical models and how the emissions are considered within different frequency ranges. The project will then move on to give detailed rundown on the computational packages used to perform the data analysis, with generated results and plots explained. Afterwards, the project investigates the importance of plasma emission modeling in gyrosynchrton emission curve fitting.

II. Theoretical background

This project investigates the radio signal data sourced from NoRP and RSTN sites, and a total of three data sequences are considered. The two sequences from RSTN is obtained from Learmonth site (apl) in Australia and Palehua site (phf) in Hawaii. The general frequency of interest lies between 0.245 GHz to 80 GHz. The three radio telescope sites are selected in a way to provide a wide converage. The three sites effectively forms a equalateral triangle, with the peak flare emission event time corresponding to the local time at each sites at around noon. Therefore, the inclusion of other sites, where the sun is not in line of sight with the telescopes would not be productive.

A. Gyrosynchrotron emissions

The primary emission is modeled as Gyrosynchrotron (gyro) emissions. It is shown that the gyro emissions can be considered as the dominating means of emissions for solar flares at relatively higher frequencies [4]. That is, when examining the emission profile of the energy released during the said flaring events, a peak emission frequency can usually be located at around 10 GHz and the emissions gradually decays onto higher frequencies [5]. For the purpose of this data analysis project, we define a low frequency threshold at 2 GHz.

A non-linear relation between the flux Φ and frequency υ can be modeled with some fit parameter A, B, a, b as follows:

$$\Phi = A \cdot \upsilon^a \cdot \left[1 - \exp\left(-B \cdot \upsilon^{-b}\right)\right] \tag{1}$$

In this simplified model, we have low frequency slope a and high frequency slope $a - b$, indicating the behaviour of gyro emissions as seen in lower and higher frequency ranges respectively. It is noted that the simplified model shown in the above equation is relatively immune to data noise [4].

B. Plasma emissions

In lower frequencies, the received radio data can be interpreted as a power law, as seen in the equation below:

$$\Phi = c \cdot \upsilon^k \tag{2}$$

Consider the general shape of a power law curve with negative k, the plasma emission profiles would dominate the lower frequencies while being relatively trivial at higher frequency ranges, where the gyro emission starts dominating. As a result, when slicing the dataset for curve fitting, we only fit the plasma model for lower frequencies, rather than using the entire data array to construct a plasma emission profile.

In theory, a partial data fit with only higher frequency dataset used for gyro emissions fitting would produce a similar result as the dataset with plasma emission model corrected. However, it is worth noticing that a complete plasma emission model along all frequency entries may provide a better adjustment to the original dataset, and consequently producing better gyro emission fits.

III. Computational methods

This project is programmed under python environment and powered by Anaconda. We use a working folder to organize the data repository, the media output, and the working python files within VSCode. In this section, we introduce the workflow and the functions within the modules. Then, we provide the overall logic of the master file, from which the generated results will be discussed in Sec. IV. The full project repository can be found on the relevant GitHub page.

A. Module, structure, and workflow

The primary library used in this project are `numpy`, `scipy`, and `matplotlib`. Seperate libraries such as `scienceplots` and `datetime` are also used to provide plotting and data conversion support.

The radio data files are stored under `./data/` in the local directory, with media output stored under `./media/`. A master script is used to get the methods constructed within other modules files. The module files are prefixed with `data_` and the master script is named `main.py`. Additionally, a separate module file called `data_legacy.py` is used to deposit legacy plotter functions.

This project employs the use of VSCode for text editing, with the modifications pushed to remote git repository using integrated bash support. Within each module file, the code blocks are constraint within separate functions, such that the master script file is able to read, assign, and pass on the variables with function calls only.

B. Data reader module

We construct a customized module, aliased `data_reader.py`, for handling initial `.csv` file content reading, formatting, and writing. During the process, specific attention is needed for the `dtype` of the given data.

For the given NoRP and RSTN dataset, the files containing time data need to be processed into `numpy.array()`. That is, by considering the given data structure, with the first column containing the number of milliseconds from the start of a day and the second column being the number of days since January 01, 1979, we employ `datetime` module to calculate and convert the two columns into a single entry along an array, where the output become a time string formatted as "YYYY-MM-DD hh:mm:ss". In effect, we loaded the time arrays as unassigned 64-bit integers (`numpy.uint64`) to avoid overflow.

C. Data handler module

1. Function: `loader()`

We start by importing the `data_reader` module into the current module. Consequently, a new function could be constructed to take in the data path of the data files and generate the formatted `numpy.array()` tuples in the return.

One thing to pay attention is that the shape of the data array must be considered to make sure that the dimensionality of the data is consistent with the other data files.

2. Function: `validator()`

Within the original dataset, a validity checker file is included to act as a filter for dealing with invalid data recordings received on the sensors. Thus, a validator function was constructed to first initiate a data mask from the validity data.

The mask is chosen such that all element on the given row must all be true when logically summed up. Afterwards, the mask is passed onto the flux and time data arrays to filter out invalid flux arrays and its corresponding time array.

3. Function: `quiet_sun()`

Another function within the `data_handler` module performs the quiet sun flux calculation. This function takes in a tuple of data arrays that requires the calculation, and generates the corresponding quiet sun flux subtracted results tuple. The operation involves in taking the mean value along a column and have all elements along the said column subtract the calculated mean value. The operation is then looped over all columns within a given `numpy.array()` and then over all arrays within the tuple.

Now, this calculation method is based on the overall readings of the flux emission over a selected period of time, that is, a longer time sequence of flux readings could provide a more accurate average flux emission value array on each frequencies, compared to an average flux value array calculated from a limited time sequence. This is due to the drastic increase in flux intensity at flare event, where the recorded flux values would be of several orders of magnitudes higher than the baseline value.

Therefore, an alternative of calculating the quiet sun flux value would be to either: 1) Obtain the annual flux readings and perform the same operations as seen above on the annual flux data; 2) To apply the negative value of the fitted gyro emission profile to the recorded flux array first, before performing the quiet sun calculations; 3) By slicing the original data array to eliminate the time sequence around the peak emission time, such that only the baseline readings are considered for quiet sun calculations.

4. Function: `collector()`

At the end, a more complex function is created to extract out, combine, and generate the combined flux and frequency array for all three dataset at the solar flare event peak time. This is done by first construct a sub-function called `peak_time()` that identifies the array row index of the peak time, from which the indexes are passed on to two other sub-functions, `peak_freq()` and `peak_flux()`, which would index out, and concatenate the arrays together into a single output.

The master function then calls to use these aforementioned sub-functions to get the combined flux and frequency array data. Notice that the frequency data array is then made unique, with non-one-to-one mapped flux data averaged. The results are then returned as a tuple.

D. Data fitter module

The `data_fitter` module utilizes `numpy`, `curve_fit` from `scipy.optimize`, and `chi2` from `scipy.stats`. A total of six functions are defined, two for fit model definition, one for getting the fitted equations' expressions for plot labeling, two more for the curve fits themselves, and one more for denoising the flux array data.

1. Function: `gyro_model()`, `plas_model()`, and `fit_label()`

The two model functions takes in the parameter arguments and return the fit functions as a direct callable. The labeling generator function unpacks the fitted results and return a `f-string` formatted entry tuple to be passed onto future plotters.

2. Function: `gyro_fitter()` and `plas_fitter()`

The two fitter functions shares similar logic in taking the flux and frequency `numpy.array()` as data. The functions run the data through the `curve_fit()` method to obtain the fit parameter tuple and its covariance matrix.

The residual term is then calculated for chi-squared statistics, where the chi-squared value is obtained with `chi2.cdf()`. Additionally, a p-value is also calculated with the degree of freedom selected from the number of data point entries and the number of fit parameters.

The two fit functions differs from results printout sections and their initial fit parameter guesses. As the power law fit function contains fewer parameters than the gyro model.

3. Function: `gyro_pass()`

The denoiser function serves to pass the combined, made unique flux and frequency data coordinates through the influence of a fitted plasma emission model in lower frequencies. That is, the fitted plasma model generates a flux data array, which would be used to adjust the original combined flux array, such that the plasma emission is effectively denoised from the original data.

The function slice the array by a cut-off threshold defined by the user at 2 GHz, pass the plasma model data adjustment, and then recombined the data arrays together to form the function return.

E. Data plotter module

A total of six figures are generated by the plotter module, with one having only valid NoRP data around the peak time, the other containing only RSTN data around the peak time. Additionally, a combined plot with NoRP and RSTN data is generated to show the flux time evolution around peak time. Then, plots containing only the peak time flux array of NoRP and RSTN is generated, along with the combined, made unique flux against frequency plot that also ships with initial `curve_fit()` results. The final plot shows a corrected flux-frequency plot with fitted curves.

By invoking the use of `scienceplots` package, we are able to globally define the plot style for `matplotlib.pyplot` such that then generated plot follows the use of LaTeXin a `Jupyter` notebook style output. For each plotter function, a log-log scale is used for the axis scale options.

1. Function: `plot_generator()`

The plot generator function takes in the tuple of all the plotting argument tuples and call their respective plotter functions. This is to reduce the module import in the master file.

IV. Analysis

In this section, we will start by providing the master python file execution logic. Then, the curve fit results are shown with generated plots.

A. Master file logic

The master script utilize the module files and import them into the script first. Before calling upon any functions, the master file defines the key variables such as local data file paths and the peak time formatted string. This step can be further optimized by passing in arguments that uses user inputs to determine the local file paths. Then, the master file calls upon the `loader()` function to deposit all the working variables into the local repository.

The filtered data is then fetched by calling upon the `validator()` function, with the quiet sun flux array calculated results deposited by invoking the `quiet_sun()` function. That is, at this stage, the master file has completed the preliminary data preparations for NoRP event analysis.

Now, the argument tuples for sub functions are created and packed into a master argument tuple, which would be passed on to the `collector()` function to generate the combined frequency and flux array at the peak radio emission time by unpacking the argument in the function call.

Effectively, the master file has now completed preparing the data arrays to be used for curve fitting. By defining a low frequency threshold, the curve fit results would be assigned to local variables by making calls on functions of `gyro_fitter()` and `plas_fitter()`. At this stage, the combined flux data array is adjusted to the fitted plasma emission model with `gyro_pass()` function call.

And finally, a tuple of plotter resources are built by rounding up the local variables. The argument tuple is then passed into `plot _generator()` to get the plots saved under local media directory.

B. Flux against frequency plots

The individual time evolution of flux data at around the purposed peak time for NoRP and RSTN sites are listed. We observe in Fig. 1 and Fig. 2 that the flux recordings around the peak time shows a consistent peak flux value at around 10 GHz, with lower frequency emissions being relatively negligible on the

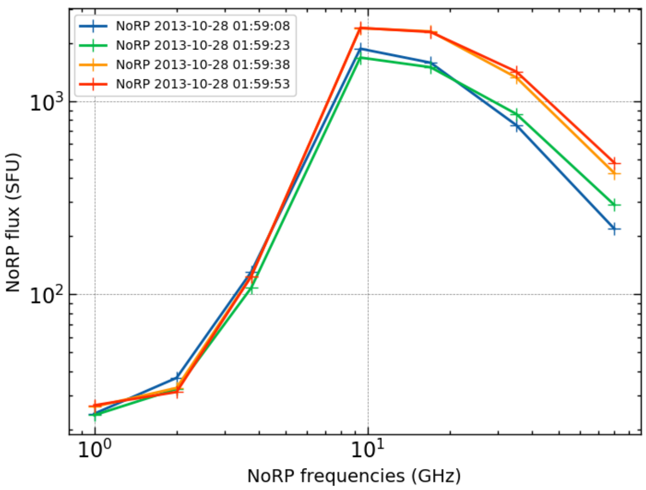

FIG. 1. Averaged flux against frequency plot for NoRP site. Flux calculated at ± 30 s around labeled time sequence. Flux sequences plotted around peak time on 2013-10-28 at 01:59:38 UTC.

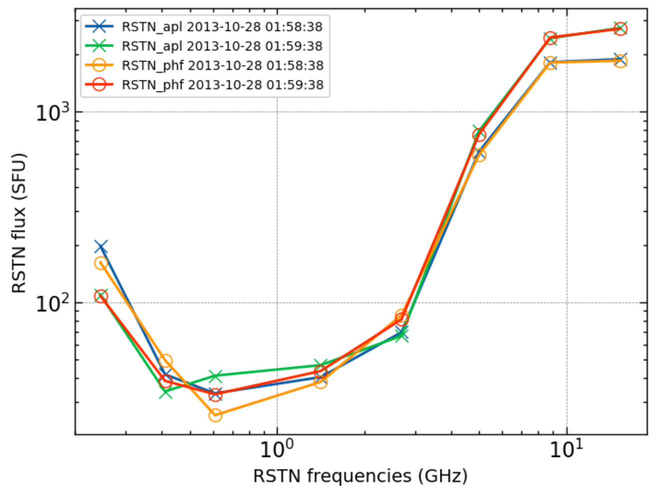

FIG. 2. Flux against frequency plot for RSTN sites. Flux calculated at $\pm$ 30 s around labeled time sequence. Flux sequences plotted around peak time on 2013-10-28 at 01:59:38 UTC.

NoRP dataset. However, it is discovered that for RSTN datasets, at lower emission frequencies, a significant amount of plasma emissions were recorded.

Additionally, plots are combined for NoRP and RSTN sites results. The combined plot as seen in Fig. 3 follows the observations from previous plots, that the data recorded from these three sites follows along with each other and forms a mutual agreement on their trends. Furthermore, the three sets of data plotted at the peak emission time suggested by NoRP data, as seen in Fig. 4 give an insight that the data set could be combined together to extend their individual frequency range.

At the end are the combined plots for NoRP and RSTN sites, the frequency and flux array were combined into a singular array respectively. The fitted curves are addressed and labeled on the plots. we now observe a clear association in the plasma emission curve fit at lower frequencies. However, the two gyro fits, both before and after the plasma emission model were considered, displays little difference in the general geometry of the fitted curve. It is worth noticing that the χ^2 test shows that, with the flux array getting adjusted with the plasma emission model, the χ^2 value decreased drastically, indicating a much better fit of the gyro model to the given combined dataset.

C. Curve fit results

The curve fitting results are printed in the terminal as listed below, the initial gyro fit with the plasma emission profile included in the data shows a low frequency slope at $a = 2.570$ and a high frequency slope at $a - b = -1.140$:

```
Gyro fitter results Initial
==============================
Gyro fitter fitted parameters
A:                      9.5675
B:                      7189.8
a:                      2.5703
b:                        3.71
Low freq slope:         2.5703
High freq slope:       -1.1397

Chi-square test result
Chi-square:         6.0352e+05
p-value:                     0
==============================
```

The initial fit generates a χ^2 value, which shows a bloated number at $\chi^2 = 6.03^5$. This provide a baseline for future fits, as higher χ^2 value, when calculated from the following method, indicates a non-optimized fit between the model and the dataset. A p-value could also be calculated. However, given the bloated χ^2 results, the associated p-value would not be of help in determining the goodness of fit in the specific case.

$$\chi^2 = \sum \left(\frac{(y_{\mathrm{data}} - y_{\mathrm{model}})^2}{y_{\mathrm{model}}} \right) \tag{3}$$

The plasma emission model as fitted for lower frequencies gives the correction model for flux data entries under 2 GHz.

```
Plas fitter results
==============================
Plas fitter fitted parameters
c:                      2.8574
k:                     -3.4994

Chi-square test result
Chi-square:             2319.9
p-value:                     0
==============================
```

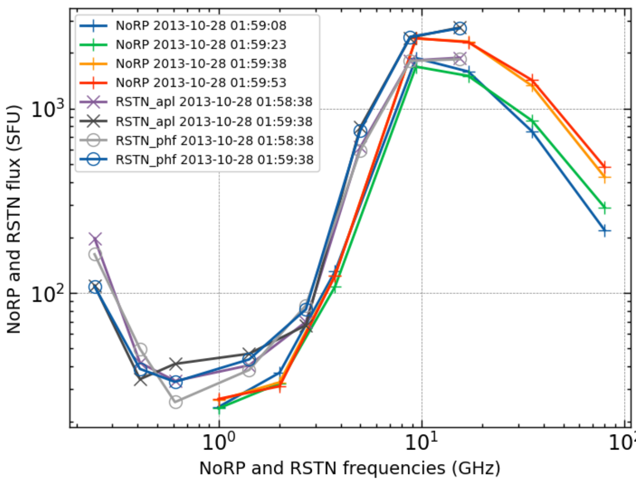

FIG. 3. Averaged flux against frequency plot for NoRP and RSTN sites. Averaged flux calculated at ± 30 s around labeled time sequence. Flux sequences plotted around peak time at 2013-10-28 at 01:59:38 UTC.

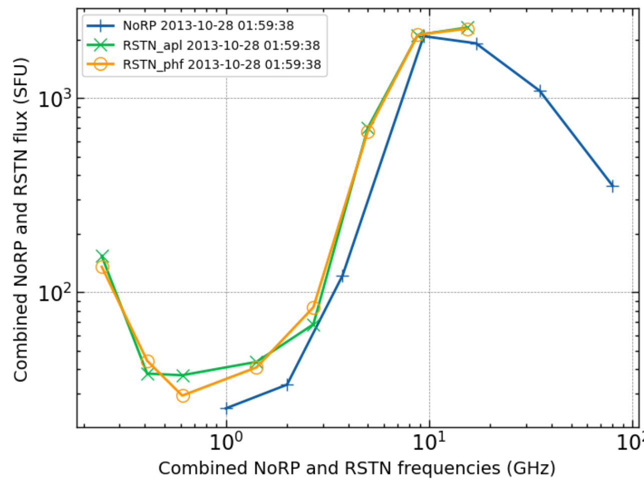

FIG. 4. Flux against frequency plot for RSTN sites. Averaged flux calculated at ± 30 s around the peak time sequence. Flux sequences plotted following the peak time at 2013-10-28 at 01:59:38 UTC.

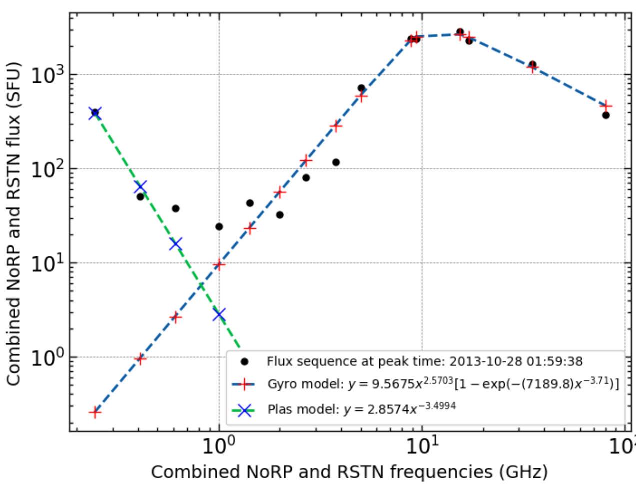

FIG. 5. Flux against frequency plot for NoRP and RSTN sites. Flux calculated at ± 30 s around labeled time sequence and combined into singular sequence. The frequencies on the frequency array are unique and maps to the flux array value one-on-one. Combined and averaged flux sequence plotted around peak time on 2013-10-28 at 01:59:38 UTC.

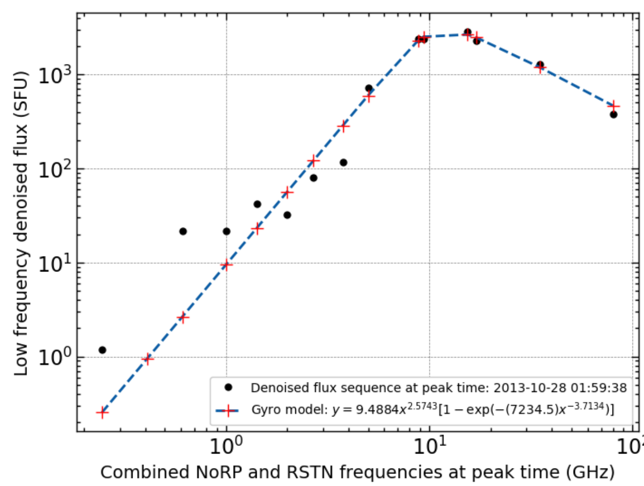

FIG. 6. Flux against frequency plot for NoRP and RSTN sites. Flux calculated at ± 30 s around labeled time sequence and combined into singular sequence. The frequencies on the frequency array are unique and maps to the flux array value one-on-one. The combined flux array is now denoised by adding the negative value of the fitted plasma emission results. Combined and averaged flux sequence plotted around peak time on 2013-10-28 at 01:59:38 UTC.

In Fig. 5, consider a plasma emission model for all frequency entries, we should expect a higher coefficient of c. By eliminating the plasma emission profile from the original combined flux curve, we refit the data and get:

```
Gyro fitter results Denoised
==============================
Gyro fitter fitted parameters
A:                      9.4884
B:                      7234.5
a:                      2.5743
b:                      3.7134
Low freq slope:         2.5743
High freq slope:       -1.1391

Chi-square test result
Chi-square:             622.28
p-value:                     0
==============================
```

The new fit illustrates a slightly different low and high frequency slope combination, with low frequency slope $a = 2.574$ and high frequency slope $a - b = -1.1391$. That is, the general trend of the fit is not affected by the inclusion of the plasma emissions model. This could be attributed to the fact that the higher flux entries are concentrated at higher frequencies, with a theorized peak at 10 GHz, effectively rendering the low frequency, plasma emission entries, as the outlier of the dataset.

However, it is worth noticing that the χ^2 value of the new fit has been reduced by a few orders of magnitude, at $\chi^2 = 6.62^2$. This indicates that the plasma emission model is still significant in providing "noise " to the gyro model fit, and denoising the data with the plasma emission could help to clean up the data to provide a better gyro fit. This is not to say that the plasma emission is generically significant to the dataset itself. However, we do recognize that the adjustment is necessary in producing better model fits. With the final fit equation at:

$$\begin{aligned}\Phi &= A \cdot \upsilon^a \cdot \left[1 - \exp\left(-B \cdot \upsilon^{-b}\right)\right] \\ &= 9.4884 \cdot \upsilon^{2.5743} \cdot \left[1 - \exp\left(-7234.5 \cdot \upsilon^{-3.7134}\right)\right]\end{aligned} \tag{4}$$

Consequently, by taking the first order derivative of the expression, we are able to discover that a peak emission frequency of the fitted gyro emission model resides around $\upsilon \simeq 12.084$ GHz. This peak emission frequency is consistent with the theory prediction at the magnitude of 10^1 GHz.

V. Conclusion

This radio data analysis projects have processed the radio data obtained from NoRP and RSTN sites regarding a peak solar flare emission event. With the generated flux vs. frequency plots and their associated curve fitting results, this project recognizes the importance of plasma emission model in its influence to gyro emissions across all frequencies of the microwave band entries. It is understood that the fitting of plasma emission model could help with denoising the received flux data array, and consequently generate better curve fitting results for gyro emissions. The computational methodology is listed and explained to provide clear guidance for repeating the data analysis steps. At the end, a non-linear least square fit of gyro emission was performed on the processed dataset, where the plasma emission profile is removed from the original dataset. The final curve fit displayed a better alignment to the dataset as seen from the χ^2 value calculations.

[1] R. Schwenn, Living Reviews in Solar Physics **3**, 2 (2006).

[2] D. N. Baker, E. Daly, I. Daglis, J. G. Kappenman, and M. Panasyuk, Space Weather **2** (2004).

[3] T. Colak and R. Qahwaji, Space Weather **7** (2009).

[4] M. Stahli, D. E. Gary, and G. J. Hurford, Solar Physics **120**, 351 (1989).

[5] K. L. Klein, Astronomy and Astrophysics **183**, 341 (1987).

A. Code file: README.md

```
## Radio data analysis
This is a course project for *ASTRO-5010* at *University of Glasgow*.
- Drafted: Mar 15, 2023
- Editted: Mar 31, 2023

## Data set
We employ the data obtained from:
1. NoRP: Nobeyama Radio Polarimeters
2. RSTN: Radio Solar Telescope Network

## Purpose
We conduct basic level data analysis and plotting to understand:
1. Gyrosynchrotron emissions
2. Plasma emissions

## Steps
We will perform the tasks with the following steps:
1. Load NoRP data set
2. Convert time array data to standard format
3. Filter out valid data readouts
4. Filter out quiet sun noise at assigned freq
5. Plot freq based flux data against time
6. Load RSTN data
7. Fit gyrosynchrotron emission
8. Fit plasma emission

## Working file structure
Main file:
- main.py

Data reader modules:
- data_reader.py

Data processing modules:
- data_handler.py

Data fitter modules:
- data_fitter.py

Plotter modules:
- data_plotter.py

## Supplementary files
Readme:
- Readme.md

Legacy plotter repo:
- data_legacy.py

## Folder structure
Data folder:
- data

Plot folder:
- media
```

B. Code file: main.py

```
"""
This is the master script of the radio data analysis project.

Created on Wed Mar 15 2023

@author: Yang-Taotao
"""
# %% Library import
# Custom module import
# Data handler import
from data_handler import loader, validator, quiet_sun, collector

# Data plotter import
from data_plotter import plot_generator, log_avg_plotter

# Data fitter import
from data_fitter import gyro_fitter, plas_fitter, gyro_pass

# %% Data path and peak time assignment
# Assign norp, apl, and phf file path
data_path = ("./data/norp/", "./data/apl/", "./data/phf/")
# Assign the peaktime of flux recording
norp_peak_time = "2013-10-28 01:59:38"

# %% Load csv data into data repo
# Deposit data arrays
data_repo = loader(data_path)
# Assign loaded data to variables
(
    norp_fi,
    norp_freq,
    norp_mvd,
    norp_tim,
    apl_fi,
    apl_freq,
    apl_tim,
    phf_fi,
    phf_freq,
    phf_tim,
) = [data_repo[i] for i in range(len(data_repo))]

# %% NoRP validity filter result deposit
norp_tim_valid, norp_fi_valid = validator(norp_mvd, norp_tim, norp_fi)

# %% NoRP quiet sun result deposit
# Generate data array tuple
quiet_sun_data = (norp_fi_valid, apl_fi, phf_fi)
# Deposit quiet sun results
norp_fi_peak, apl_fi_peak, phf_fi_peak = quiet_sun(quiet_sun_data)

# %% Peak time array argument assignment
# Time, Freq, Flux sub function arguemnt repo
arg_time, arg_freq, arg_flux = (
    # arg_time
    (
        norp_tim_valid,
```

```
        apl_tim,
        phf_tim,
        norp_peak_time,
    ),
    # arg_freq
    (
        norp_freq,
        apl_freq,
        phf_freq,
    ),
    # arg_flux
    (
        norp_fi_peak,
        apl_fi_peak,
        phf_fi_peak,
    ),
)
# Peak time arguement tally
peak_arg = (arg_time, arg_freq, arg_flux)

# %% Get peak time combined flux and time array
peak_time_freq, peak_time_flux = collector(*peak_arg)

# %% Curve fitter key value assignment - Define cut-off freq
freq_cut = 2

# %% Generate combined fit results
# Get fitted results
results_gyro, results_plas = (
    gyro_fitter(peak_time_freq, peak_time_flux, "Initial"),
    plas_fitter(peak_time_freq, peak_time_flux, freq_cut),
)
# Assign fit parameters
gyro_param, plas_param = results_gyro[0], results_plas[0]

# %% Flux array denoise at low freq
gyro_freq_denoise, gyro_flux_denoise = (
    peak_time_freq,
    gyro_pass(peak_time_freq, peak_time_flux, freq_cut, plas_param),
)

# %% Refit with denoised data
results_denoise = gyro_fitter(gyro_freq_denoise, gyro_flux_denoise, "Denoised")
denoise_param = results_denoise[0]

# %% Plotter argument assignment
# Plot arguments assignment
plt_arg1, plt_arg2, plt_arg3, plt_arg4, plt_arg5 = (
    # NoRP plotter arguments
    (norp_tim_valid, norp_fi_peak, norp_freq, norp_peak_time),
    # RSTN plotter arguments
    (
        apl_tim,
        phf_tim,
        apl_fi_peak,
        phf_fi_peak,
        apl_freq,
        phf_freq,
```

```
        norp_peak_time,
    ),
    # Combined plotter arguments
    (
        norp_tim_valid,
        apl_tim,
        phf_tim,
        norp_fi_peak,
        apl_fi_peak,
        phf_fi_peak,
        norp_freq,
        apl_freq,
        phf_freq,
        norp_peak_time,
    ),
    # Peak time combined plotter arguments
    (
        peak_time_freq,
        peak_time_flux,
        norp_peak_time,
        gyro_param,
        plas_param,
        freq_cut,
    ),
    # Denoised plotter arguments
    (
        gyro_freq_denoise,
        gyro_flux_denoise,
        norp_peak_time,
        denoise_param,
        plas_param,
    ),
)

# %% Optional - Averaged flux array parser
norp_peak_avg, apl_peak_avg, phf_peak_avg = log_avg_plotter(plt_arg3)

# %% Plot generator argument assignment
plt_arg = (plt_arg1, plt_arg2, plt_arg3, plt_arg4, plt_arg5)

# %% Plot generation
plot_generator(plt_arg)
```

C. Code file: data_reader.py

```python
"""
This is the data loader script of the radio data analysis project.

Created on Wed Mar 15 2023

@author: Yang-Taotao
"""
# %% Library import
# Library import
import datetime as dt
import numpy as np

# %% Data parser
# CSV data parser
def csv_loader(file_path, dtype=float):
    """
    Parameters
    ----------
    file_path : string
        Path to data file folder.
    dtype : dtype, optional
        The dtype of assigned file. The default is float.

    Returns
    -------
    data : array
        The data readout array.

    """
    # Initial CSV data load with specified dtype
    data = np.loadtxt(file_path, delimiter=",", dtype=dtype)

    # Time data loader and convertor
    if dtype == np.uint64:
        # Define start point of datetime at 1979-01-01 as day01
        time_origin = dt.datetime(1979, 1, 1) - dt.timedelta(days=1)

        # Construct time function for calculating time result
        time_update = [
            time_origin + dt.timedelta(milliseconds=ms + days * 86400000)
            for ms, days in data  # col0, col1: ms, days
        ]

        # Combined datetime data
        data = np.array(
            [time.strftime("%Y-%m-%d %H:%M:%S") for time in time_update]
        )

        # Return converted time data array
        return data

    # Data loader for all other dtype
    return data
```

D. Code file: data_handler.py

```python
"""
This is the data handler script of the radio data analysis project.

Created on Wed Mar 15 2023

@author: Yang-Taotao
"""
# %% Library import
# Library import
import numpy as np
from data_reader import csv_loader

# %% Data loader
# Combined data loader
def loader(data_path):
    """
    Parameters
    ----------
    data_path : tuple
        Tuple of data folder path.

    Returns
    -------
    result : tuple
        Tuple of loaded data arrays.
    """
    # Local path variable repo
    flux, freq, mvd, tim = ("flux.csv", "freq.csv", "mvd.csv", "tim.csv")

    # Pack data into tuple
    result = (
        csv_loader(data_path[0] + flux),
        csv_loader(data_path[0] + freq),
        csv_loader(data_path[0] + mvd, dtype=int),
        csv_loader(data_path[0] + tim, dtype=np.uint64),
        csv_loader(data_path[1] + flux).transpose(),
        csv_loader(data_path[1] + freq),
        csv_loader(data_path[1] + tim, dtype=np.uint64),
        csv_loader(data_path[2] + flux).transpose(),
        csv_loader(data_path[2] + freq),
        csv_loader(data_path[2] + tim, dtype=np.uint64),
    )

    # Return the assignment
    return result


# %% Data validator
# NORP data filter based on mvd file
def validator(data_norp_mvd, data_norp_tim, data_norp_fi):
    """
    Parameters
    ----------
    data_norp_mvd : array
        Validity array of NORP data.
    data_norp_tim : array
```

```
        Time array of NORP data.
    data_norp_fi : array
        Flux array of NORP data.

    Returns
    -------
    data_norp_tim_valid : array
        Valid time array of NORP data.
    data_norp_fi_valid : array
        Valid flux array of NORP data.
    """
    # Generate valid data mask based on boolean readout over single rows
    data_norp_mask = np.all(data_norp_mvd.astype(bool), axis=1)

    # Filter the time and flux data through mask
    data_norp_tim_valid, data_norp_fi_valid = (
        data_norp_tim[data_norp_mask],
        data_norp_fi[data_norp_mask],
    )

    # Return filtered result
    return (data_norp_tim_valid, data_norp_fi_valid)


# %% Quiet sun flux calculator
# Quiet sun calculator
def quiet_sun(data_array_tuple):
    """
    Parameters
    ----------
    data_array_tuple : tuple
        Flux array tuple.

    Returns
    -------
    data_array_repo : tuple
        Filtered quiet sun array data tuple.
    """
    # Loop through the arrays to generate quiet sun flux array tuple
    data_array_repo = tuple(
        array - np.mean(array, axis=0) for array in data_array_tuple
    )

    # Return quiet sun flux array tuple
    return data_array_repo


# %% Peak time array collector
# Peak time array collector
def collector(arg_time, arg_freq, arg_flux):
    """
    Parameters
    ----------
    arg_time : tuple
        Tuple of time and peak time data arrays.
    arg_freq : tuple
        Tuple of freq data arrays.
    arg_flux : tuple
```

```
        Tuple of flux data arrays.

    Returns
    -------
    data_fi_combined : array
        Peak time flux array combined.
    data_freq_combined : array
        Peak time freq array combined.
    """
    # Peak time index identifier and freq combiner
    def peak_time(arg_time):
        """
        Parameters
        ----------
        arg : tuple
            Tuple of time data arrays with peak time.

        Returns
        -------
        idx_norp : integer
            Index of NoRP peak time.
        idex_apl : integer
            Index of apl peak time.
        idx_phf : integer
            Index of phf peak time.
        data_freq_peak_time_combined : array
            Combined freq array.
        """
        # Local variable repo
        (
            data_norp_tim_valid,
            data_apl_tim,
            data_phf_tim,
            data_norp_peak_time,
        ) = [arg_time[i] for i in range(len(arg_time))]

        # Index locator
        idx_norp, idx_apl, idx_phf = (
            np.where(data_norp_tim_valid == data_norp_peak_time)[0][0],
            np.where(data_apl_tim == data_norp_peak_time)[0][0],
            np.where(data_phf_tim == data_norp_peak_time)[0][0],
        )

        # Return index repo
        return (idx_norp, idx_apl, idx_phf)

    # Peak time freq collector
    def peak_freq(arg_freq):
        """
        Parameters
        ----------
        arg_freq : tuple
            Tuple of freq data arrays.

        Returns
        -------
        data_freq_combined : array
            Combined freq array.
```

```
        """
        # Local variable repo
        data_norp_freq, data_apl_freq, data_phf_freq = [
            arg_freq[i] for i in range(len(arg_freq))
        ]

        # Concatenate freq array
        data_freq = tuple(
            [
                data_norp_freq,
                data_apl_freq,
                data_phf_freq,
            ]
        )
        data_freq_combined = np.concatenate(data_freq)

        # Return combined freq array
        return data_freq_combined

    # Peak time flux collector
    def peak_flux(arg_flux):
        """
        Parameters
        ----------
        arg_flux : tuple
            Tuple of flux data arrays.

        Returns
        -------
        data_flux_combined : array
            Combined flux array.

        """
        # Local variable repo
        data_norp_flux_peak, data_apl_flux_peak, data_phf_flux_peak = [
            arg_flux[i] for i in range(len(arg_flux))
        ]

        # Import peak identifier result
        idx_norp, idx_apl, idx_phf = peak_time(arg_time)

        # Pick peak time flux array out
        data_norp_flux_peak, data_apl_flux_peak, data_phf_flux_peak = (
            data_norp_flux_peak[idx_norp],
            data_apl_flux_peak[idx_apl],
            data_phf_flux_peak[idx_phf],
        )

        # Concatenate freq array
        data_flux = tuple(
            [
                data_norp_flux_peak,
                data_apl_flux_peak,
                data_phf_flux_peak,
            ]
        )
        data_flux_combined = np.concatenate(data_flux)

```

```
        # Return combined freq array
        return data_flux_combined

    # Peak time freq, flux array generator
    data_freq_combined, data_flux_combined = peak_freq(arg_freq), peak_flux(
        arg_flux
    )

    # Array sorter
    # Get numpy index sort array
    idx_sort = np.argsort(data_freq_combined)
    # Sort array with index
    data_freq_sorted, data_flux_sorted = (
        data_freq_combined[idx_sort],
        data_flux_combined[idx_sort],
    )

    # Array uniqe value filter
    # Get the unique freq values array
    data_freq_final = np.unique(data_freq_sorted)

    # Calculate the mean y values for each unique x value
    data_flux_final = np.array(
        [
            np.mean(data_flux_sorted[data_freq_sorted == xval])
            for xval in data_freq_final
        ]
    )

    # Return combined peak time flux array
    return (data_freq_final, data_flux_final)
```

E. Code file: data_fitter.py

```
"""
This is the data fitter script of the radio data analysis project.

Created on Wed Mar 15 2023

@author: Yang-Taotao
"""
# %% Library import
# Library import
import numpy as np
from scipy.optimize import curve_fit
from scipy.stats import chi2

# %% Fit model definition
# Gyro model definition
def gyro_model(x_val, param_a_cap, param_b_cap, param_a, param_b):
    """
    Parameters
    ----------
    x_val : array
        Gyro model x value array.
    param_a_cap : float
        Fit param A.
    param_b_cap : float
        Fit param B.
    param_a : float
        Fit param a.
    param_b : float
        Fit param b.

    Returns
    -------
    array
        Gyro model.
    """
    # Return gyro model
    return (
        param_a_cap
        * (x_val**param_a)
        * (1 - np.exp(-param_b_cap * (x_val ** (-param_b))))
    )


# Plas model definition
def plas_model(x_val, param_c, param_k):
    """
    Parameters
    ----------
    x_val : array
        Plasma model x value array.
    param_c : float
        Fit param c.
    param_k : float
        Fit param k.

    Returns
```

```
    -------
    array
        Plasma model.

    """
    # Return plas model
    return param_c * (x_val**param_k)


# %% Fitted function result label generator
def fit_label(gyro_param, plas_param):
    """
    Parameters
    ----------
    gyro_param : tuple
        Gyro model fit param tuple.
    plas_param : tuple
        Plas model fit param tuple.

    Returns
    -------
    label_gyro : string
        Fitted gyro model expression.
    label_plas : string
        Fitted plas model expression.
    """
    # Cache into singular tuple
    fit_param = np.concatenate([gyro_param, plas_param]).ravel()

    # Local fit cariable unpack
    fit_a_cap, fit_b_cap, fit_a, fit_b, fit_c, fit_k = [
        fit_param[i] for i in range(len(fit_param))
    ]

    # Label generator
    label_gyro, label_plas = (
        # Gyro model
        rf"Gyro model: $y={fit_a_cap:.5g}x^{{{fit_a:.5g}}}"
        rf"[1-\exp({{-({fit_b_cap:.5g})x^{{-{fit_b:.5g}}}}})]$",
        # Plas model
        rf"Plas model: $y={fit_c:.5g}x^{{{fit_k:.5g}}}$",
    )

    # Result return
    return (label_gyro, label_plas)


# %% Gyro fitter
# Gyro fitter function
def gyro_fitter(data_freq, data_flux, title):
    """
    Parameters
    ----------
    data_freq : array
        Combined freq data array.
    data_flux : array
        Combined flux data array.
    title : string
```

```
        Additional title for plot customization.

    Returns
    -------
    params : array
        Fit parameters array.
    cov : float
        Covariance matrix of the fit.
    chi_sqr :float
        Chi-squared value of the fit.
    chi_p_val : float
        The p-value from the chi-sqaured test.
    """
    # Generate filtered data
    data_x, data_y = (
        data_freq,
        data_flux,
    )

    # Iniitial parameter guess
    param_guess = [1, 1, 1, 1]  # param_A, param_B, param_a, param_b

    # Curve fit results
    params, cov = curve_fit(gyro_model, data_x, data_y, p0=param_guess)

    # Residuals generator
    # Get fitted model
    fit_model = gyro_model(data_x, *params)
    # Residual generator
    fit_resid = data_y - fit_model

    # Chi2 Tester
    # Chi2 calculation and dof generation
    chi_sqr, chi_dof = (
        np.sum(fit_resid**2 / fit_model),
        len(data_x) - len(params),
    )
    # Chi2 p-value calculation
    chi_p_val = 1 - chi2.cdf(chi_sqr, chi_dof)

    # Results print out
    # Gyro fitter result title
    print()
    print(f"{'Gyro fitter results ' + title:<20}")
    print("=" * 30)
    # Print fit parameters
    print(f"{'Gyro fitter fitted parameters':<15}")
    print(f"{'A:':<20}{params[0]:>10.5g}")
    print(f"{'B:':<20}{params[1]:>10.5g}")
    print(f"{'a:':<20}{params[2]:>10.5g}")
    print(f"{'b:':<20}{params[3]:>10.5g}")
    print(f"{'Low freq slope:':<20}{params[2]:>10.5g}")
    print(f"{'High freq slope:':<20}{params[2]-params[3]:>10.5g}")
    print()
    # Print chi2 results
    print(f"{'Chi-square test result':<20}")
    print(f"{'Chi-square:':<20}{chi_sqr:>10.5g}")
    print(f"{'p-value:':<20}{chi_p_val:>10.5g}")
```

```python
    print("=" * 30)
    print()

    # Function return
    return (params, cov, chi_sqr, chi_p_val)


# %% Plas fitter
# Plas fitter function
def plas_fitter(data_x, data_y, cut):
    """
    Parameters
    ----------
    data_x : array
        Combined freq data array.
    data_y : array
        Combined flux data array.
    cut : float
        Cut-off point for different fits.

    Returns
    -------
    params : array
        Fit parameters array.
    cov : float
        Covariance matrix of the fit.
    chi_sqr :float
        Chi-squared value of the fit.
    chi_p_val : float
        The p-value from the chi-sqaured test.
    """
    # Generate filtered data
    data_x, data_y = (
        data_x[data_x < cut],
        data_y[data_x < cut],
    )

    # Iniitial parameter guess
    param_guess = [1, 1]  # param_c, param_k

    # Curve fit results
    params, cov = curve_fit(plas_model, data_x, data_y, p0=param_guess)

    # Residuals generator
    # Get fitted model
    fit_model = plas_model(data_x, *params)
    # Residual generator
    fit_resid = data_y - fit_model

    # Chi2 Tester
    # Chi2 calculation and dof generation
    chi_sqr, chi_dof = (
        np.sum((fit_resid) ** 2 / fit_model),
        len(data_x) - len(params),
    )
    # Chi2 p-value calculation
    chi_p_val = 1 - chi2.cdf(chi_sqr, chi_dof)

```

```
    # Results print out
    # Gyro fitter result title
    print()
    print(f"{'Plas fitter results':<20}")
    print("=" * 30)
    # Print fit parameters
    print(f"{'Plas fitter fitted parameters':<20}")
    print(f"{'c:':<20}{params[0]:>10.5g}")
    print(f"{'k:':<20}{params[1]:>10.5g}")
    print()
    # Print chi2 results
    print(f"{'Chi-square test result':<20}")
    print(f"{'Chi-square:':<20}{chi_sqr:>10.5g}")
    print(f"{'p-value:':<20}{chi_p_val:>10.5g}")
    print("=" * 30)
    print()

    # Function return
    return (params, cov, chi_sqr, chi_p_val)


# %% Gyro model denoiser (cut plas model)
def gyro_pass(data_freq, data_flux, cut, plas_param):
    """
    Parameters
    ----------
    data_freq : array
        Peak time freq array.
    data_flux : array
        Peak time flux array.
    cut : float
        Low freq cut-off value.
    plas_param : tuple
        Tuple of plas model fit param.

    Returns
    -------
    data_y_gyro : array
        Denoised flux array for gyro fitting.
    """
    # Cut-off array local repo
    data_x, data_y, data_y_base = (
        data_freq[data_freq < cut],
        data_flux[data_freq < cut],
        data_flux[data_freq >= cut],
    )

    # Get denoised flux array in plas model domain
    data_y_pass = data_y - plas_model(data_x, *plas_param)

    # Get denoised flux data array for gyro fitting
    data_y_gyro = np.concatenate([data_y_pass, data_y_base])

    # Return denoised result
    return data_y_gyro
```

F. Code file: data_plotter.py

```python
"""
This is the data plotter script of the radio data analysis project.

Created on Wed Mar 15 2023

@author: Yang-Taotao
"""
# %% Library import
# Library import
import numpy as np
import matplotlib.pyplot as plt
import scienceplots

# Custom module import
from data_fitter import gyro_model, plas_model, fit_label

# %%  Plot style config
# Plot style configuration
plt.style.use(["science", "notebook", "grid"])

# %% NoRP plotter
# NoRP log log plotter - flux vs freq at each time - 100 index = 10 s
def norp_plotter(arg):
    """
    Parameters
    ----------
    arg : tuple
        Plotter argument parameters.

    Returns
    -------
    None.
    """
    # Local variable repo
    (
        data_norp_tim_valid,
        data_norp_fi_peak,
        data_norp_freq,
        data_norp_peak_time,
    ) = [arg[i] for i in range(len(arg))]

    # Plot range limiter
    # Gain peak value time array index
    peak = (
        np.where(data_norp_tim_valid == data_norp_peak_time)[0][0],  # peak
        300,  # peak_gap
        150,  # gap
    )
    # Plot data range limiter at +- 30s --> peak_gap >= 300
    # Structure - (peak_start, peak_end)
    peak_idx = (
        max(0, peak[0] - peak[1]),  # peak_start
        min(data_norp_fi_peak.shape[0], peak[0] + peak[1]),  # peak_end
    )

    # Plot with loops
```

```python
    plt_0 = [
        plt.plot(
            data_norp_freq,
            np.mean(
                data_norp_fi_peak[i : i + peak[2]], axis=0
            ),  # Mean data calculator
            "+-",
            markersize=10,
            label="NoRP " + data_norp_tim_valid[i],
        )
        for i in range(peak_idx[0], peak_idx[1], peak[2])
    ]

    # Plot axis scale definer
    plt.xscale("log")
    plt.yscale("log")

    # Plot customizations
    plt.xlabel("NoRP frequencies (GHz)", fontsize=14)
    plt.ylabel("NoRP flux (SFU)", fontsize=14)
    plt.legend(fontsize=10)

    # Save and close
    plt.savefig("./media/figure_01_norp.png")
    plt.close()

    # Return function call
    return plt_0


# %% RSTN plotter
# RSTN log log plotter - flux vs freq at each time - 200 index = 190 s
def rstn_plotter(arg):
    """
    Parameters
    ----------
    arg : tuple
        Plotter argument parameters.

    Returns
    -------
    None.
    """
    # Local variable repo
    (
        data_apl_tim,
        data_phf_tim,
        data_apl_fi_peak,
        data_phf_fi_peak,
        data_apl_freq,
        data_phf_freq,
        data_norp_peak_time,
    ) = [arg[i] for i in range(len(arg))]

    # Plot range limiter
    # Gain peak value time array index
    peak = (
        np.where(data_apl_tim == data_norp_peak_time)[0][0],  # peak_apl
```

```
            np.where(data_phf_tim == data_norp_peak_time)[0][0],  # peak_phf
            60,  # peak_gap
            60,  # gap
        )
        # Plot data range limiter at +- 30s
        peak_idx = (
            max(0, peak[0] - peak[2]),  # peak_apl_start
            min(data_apl_fi_peak.shape[0], peak[0] + peak[2]),  # peak_apl_end
            max(0, peak[1] - peak[2]),  # peak_phf_start
            min(data_phf_fi_peak.shape[0], peak[1] + peak[2]),  # peak_phf_end
        )

        # Plot generation
        # Plot apl with loops
        plt_0 = [
            plt.plot(
                data_apl_freq,
                np.mean(
                    data_apl_fi_peak[i : i + peak[3]], axis=0
                ),  # Mean data calculator
                "x-",
                markersize=10,
                label="RSTN_apl " + data_apl_tim[i],
            )
            for i in range(peak_idx[0], peak_idx[1], peak[3])
        ]
        # Plot phf with loops
        plt_1 = [
            plt.plot(
                data_phf_freq,
                np.mean(
                    data_phf_fi_peak[i : i + peak[3]], axis=0
                ),  # Mean data calculator
                "o-",
                markerfacecolor="none",
                markersize=10,
                label="RSTN_phf " + data_phf_tim[i],
            )
            for i in range(peak_idx[2], peak_idx[3], peak[3])
        ]

        # Plot axis scale definer
        plt.xscale("log")
        plt.yscale("log")

        # Plot customizations
        plt.xlabel("RSTN frequencies (GHz)", fontsize=14)
        plt.ylabel("RSTN flux (SFU)", fontsize=14)
        plt.legend(fontsize=10)

        # Save and close
        plt.savefig("./media/figure_02_rstn.png")
        plt.close()

        # Return function call
        return (plt_0, plt_1)
```

```python
# %% Combined plotter
# Combined plotter
def log_plotter(arg):
    """
    Parameters
    ----------
    arg : tuple
        Plotter argument parameters.

    Returns
    -------
    None.
    """
    # Local variable repo
    (
        data_norp_tim_valid,
        data_apl_tim,
        data_phf_tim,
        data_norp_fi_peak,
        data_apl_fi_peak,
        data_phf_fi_peak,
        data_norp_freq,
        data_apl_freq,
        data_phf_freq,
        data_norp_peak_time,
    ) = [arg[i] for i in range(len(arg))]

    # Plot range limiter
    # Gain peak value time array index
    peak = (
        np.where(data_norp_tim_valid == data_norp_peak_time)[0][0],  # peak
        np.where(data_apl_tim == data_norp_peak_time)[0][0],  # peak_apl
        np.where(data_phf_tim == data_norp_peak_time)[0][0],  # peak_phf
        300,  # peak_norp_gap
        60,  # peak_rstn_gap
        150,  # gap_norp
        60,  # gap_rstn
    )
    # Plot data range limiter at +- 30s
    peak_idx = (
        max(0, peak[0] - peak[3]),  # peak_start
        min(data_norp_fi_peak.shape[0], peak[0] + peak[3]),  # peak_end
        max(0, peak[1] - peak[4]),  # peak_apl_start
        min(data_apl_fi_peak.shape[0], peak[1] + peak[4]),  # peak_apl_end
        max(0, peak[2] - peak[4]),  # peak_phf_start
        min(data_phf_fi_peak.shape[0], peak[2] + peak[4]),  # peak_phf_end
    )

    # Plot generation
    # Plot norp with loops
    plt_0 = [
        plt.plot(
            data_norp_freq,
            np.mean(
                data_norp_fi_peak[i : i + peak[5]], axis=0
            ),  # Mean data calculator
            "+-",
            markersize=10,
```

```
                label="NoRP " + data_norp_tim_valid[i],
            )
            for i in range(peak_idx[0], peak_idx[1], peak[5])
        ]
        # Plot apl with loops
        plt_1 = [
            plt.plot(
                data_apl_freq,
                np.mean(
                    data_apl_fi_peak[i : i + peak[6]], axis=0
                ),  # Mean data calculator
                "x-",
                markersize=10,
                label="RSTN_apl " + data_apl_tim[i],
            )
            for i in range(peak_idx[2], peak_idx[3], peak[6])
        ]
        # Plot phf with loops
        plt_2 = [
            plt.plot(
                data_phf_freq,
                np.mean(
                    data_phf_fi_peak[i : i + peak[6]], axis=0
                ),  # Mean data calculator
                "o-",
                markerfacecolor="none",
                markersize=10,
                label="RSTN_phf " + data_phf_tim[i],
            )
            for i in range(peak_idx[4], peak_idx[5], peak[6])
        ]

        # Plot axis scale definer
        plt.xscale("log")
        plt.yscale("log")

        # Plot customizations
        plt.xlabel("NoRP and RSTN frequencies (GHz)", fontsize=14)
        plt.ylabel("NoRP and RSTN flux (SFU)", fontsize=14)
        plt.legend(fontsize=10)

        # Save and close
        plt.savefig("./media/figure_03_combined.png")
        plt.close()

        # Return function call
        return (plt_0, plt_1, plt_2)


# %% Combined peak time plotter
# Combined peak time plotter
def log_avg_plotter(arg):
    """
    Parameters
    ----------
    arg : tuple
        Plotter argument parameters.

```

```python
    Returns
    -------
    None.
    """
    # Local variable repo
    (
        data_norp_tim_valid,
        data_apl_tim,
        data_phf_tim,
        data_norp_fi_peak,
        data_apl_fi_peak,
        data_phf_fi_peak,
        data_norp_freq,
        data_apl_freq,
        data_phf_freq,
        data_norp_peak_time,
    ) = [arg[i] for i in range(len(arg))]

    # Plot range limiter
    # Gain peak value time array index
    peak = (
        np.where(data_norp_tim_valid == data_norp_peak_time)[0][0],  # peak
        np.where(data_apl_tim == data_norp_peak_time)[0][0],  # peak_apl
        np.where(data_phf_tim == data_norp_peak_time)[0][0],  # peak_phf
        300,  # peak_norp_gap
        60,  # peak_rstn_gap
        150,  # gap_norp
        60,  # gap_rstn
    )
    # Plot data range limiter at +- 30s
    peak_idx = (
        max(0, peak[0] - peak[3]),  # peak_start
        min(data_norp_fi_peak.shape[0], peak[0] + peak[3]),  # peak_end
        max(0, peak[1] - peak[4]),  # peak_apl_start
        min(data_apl_fi_peak.shape[0], peak[1] + peak[4]),  # peak_apl_end
        max(0, peak[2] - peak[4]),  # peak_phf_start
        min(data_phf_fi_peak.shape[0], peak[2] + peak[4]),  # peak_phf_end
    )

    # Generate peak time averaged data array
    data_norp_peak_avg, data_apl_peak_avg, data_phf_peak_avg = (
        data_norp_fi_peak[peak_idx[0] : peak_idx[1], :],
        data_apl_fi_peak[peak_idx[2] : peak_idx[3], :],
        data_phf_fi_peak[peak_idx[4] : peak_idx[5], :],
    )

    # Plot generation
    # Plot norp with loops
    plt.plot(
        data_norp_freq,
        np.mean(data_norp_peak_avg, axis=0),
        "+-",
        markersize=10,
        label="NoRP " + data_norp_tim_valid[peak[0]],
    )
    # Plot apl with loops
    plt.plot(
        data_apl_freq,
```

```
        np.mean(data_apl_peak_avg, axis=0),
        "x-",
        markersize=10,
        label="RSTN_apl " + data_apl_tim[peak[1]],
    )
    # Plot phf with loops
    plt.plot(
        data_phf_freq,
        np.mean(data_phf_peak_avg, axis=0),
        "o-",
        markerfacecolor="none",
        markersize=10,
        label="RSTN_phf " + data_phf_tim[peak[2]],
    )

    # Plot axis scale definer
    plt.xscale("log")
    plt.yscale("log")

    # Plot customizations
    plt.xlabel("Combined NoRP and RSTN frequencies (GHz)", fontsize=14)
    plt.ylabel("Combined NoRP and RSTN flux (SFU)", fontsize=14)
    plt.legend(fontsize=10)

    # Save and close
    plt.savefig("./media/figure_03_peak_average_combined.png")
    plt.close()

    # Return averaged flux array at peak time
    return (data_norp_peak_avg, data_apl_peak_avg, data_phf_peak_avg)


# %% Peak plotter
# Peak time plotter
def peak_plotter(arg):
    """
    Parameters
    ----------
    arg : tuple
        Peak plotter argument with freq, flux, and peak time.

    Returns
    -------
    None.

    """
    # Local variable repo
    (
        data_peak_freq,
        data_peak_flux,
        data_norp_peak_time,
        gyro_param,
        plas_param,
        freq_cut,
    ) = [arg[i] for i in range(len(arg))]

    # Fitted function repo
    data_gyro, data_plas = (
```

```
        gyro_model(data_peak_freq, *gyro_param),
        plas_model(data_peak_freq[data_peak_freq < freq_cut], *plas_param),
    )

    # Fit label local repo
    label_gyro, label_plas = fit_label(gyro_param, plas_param)

    # Plot generation
    plt_0 = plt.plot(
        data_peak_freq,
        data_peak_flux,
        ".",
        color="black",
        markersize=10,
        markeredgecolor="black",
        label="Flux sequence at peak time: " + data_norp_peak_time,
    )
    # Gyro fit curve - x >= 2
    plt_1 = plt.plot(
        data_peak_freq,
        data_gyro,
        "+--",
        markersize=10,
        markeredgecolor="red",
        label=label_gyro,
    )
    # Plas fit curve - x < 2
    plt_2 = plt.plot(
        data_peak_freq[data_peak_freq < freq_cut],
        data_plas,
        "x--",
        markersize=10,
        markeredgecolor="blue",
        label=label_plas,
    )

    # Plot axis scale definer
    plt.xscale("log")
    plt.yscale("log")

    # Plot customizations
    plt.xlabel("Combined NoRP and RSTN frequencies (GHz)", fontsize=14)
    plt.ylabel("Combined NoRP and RSTN flux (SFU)", fontsize=14)
    plt.legend(fontsize=10)
    plt.savefig("./media/figure_03_peak_time.png")
    plt.close()

    # Return function call
    return (plt_0, plt_1, plt_2)


# %% Denoised data peak time plotter
def denoise_plotter(arg):
    # Local variable repo
    (
        data_peak_freq,
        data_peak_flux_denoise,
        data_norp_peak_time,
```

```
        denoise_param,
        plas_param,
    ) = [arg[i] for i in range(len(arg))]

    # Fitted gyro function repo
    data_gyro = gyro_model(data_peak_freq, *denoise_param)

    # Fit label local repo
    label_gyro = fit_label(denoise_param, plas_param)[0]

    # Plot generation
    plt_0 = plt.plot(
        data_peak_freq,
        data_peak_flux_denoise,
        ".",
        color="black",
        markersize=10,
        label="Denoised flux sequence at peak time: " + data_norp_peak_time,
    )
    # Gyro fit curve
    plt_1 = plt.plot(
        data_peak_freq,
        data_gyro,
        "+--",
        markersize=10,
        markeredgecolor="red",
        label=label_gyro,
    )

    # Plot axis scale definer
    plt.xscale("log")
    plt.yscale("log")

    # Plot customizations
    plt.xlabel(
        "Combined NoRP and RSTN frequencies at peak time (GHz)", fontsize=14
    )
    plt.ylabel("Low frequency denoised flux (SFU)", fontsize=14)
    plt.legend(fontsize=10)
    plt.savefig("./media/figure_04_denoised.png")
    plt.close()

    # Return function call
    return (plt_0, plt_1)


# %% Plot generator
# Define generator function
def plot_generator(arg):
    """
    Parameters
    ----------
    arg : array
        Plot generator argument with sub plotter arguments.

    Returns
    -------
    results : None
```

```
        A collection of plots.
    """
    # Local variable repo
    arg_norp, arg_rstn, arg_combine, arg_peak, arg_denoise = [
        arg[i] for i in range(len(arg))
    ]

    # Result compilation
    results = (
        norp_plotter(arg_norp),
        rstn_plotter(arg_rstn),
        log_plotter(arg_combine),
        log_avg_plotter(arg_combine),
        peak_plotter(arg_peak),
        denoise_plotter(arg_denoise),
    )

    # Return function call
    return results
```

G. Code file: data_legacy.py

```python
"""
This is the data plotter legacy script of the radio data analysis project.

Created on Tue Mar 28 2023

@author: Yang-Taotao
"""
# Library import
import numpy as np
import matplotlib.pyplot as plt
import scienceplots

# Custom module import
from data_fitter import gyro_model, plas_model

# Plot style configuration
plt.style.use(["science", "notebook", "grid"])

# Plotters
# NORP plotter - flux vs time at each freq
def norp_plotter(
    data_norp_tim_valid, data_norp_fi_peak, data_norp_peak_time, data_norp_freq
):
    # Plot with loops
    [
        plt.plot(
            data_norp_tim_valid,
            data_norp_fi_peak[:, i],
            label=data_norp_freq[i],
        )
        for i in range(data_norp_fi_peak.shape[1])
    ]

    # Plotting for the peak value
    plt.axvline(x=data_norp_peak_time, color="crimson", linestyle="--")

    # Gain peak value x-axis index
    peak = np.where(data_norp_tim_valid == data_norp_peak_time)[0][0]
    x_lim_left, x_lim_right = (
        data_norp_tim_valid[peak - 2000],
        data_norp_tim_valid[peak + 2000],
    )

    # Ticks declutter
    plt.xticks(data_norp_tim_valid[::500], rotation=90, fontsize=10)

    # Plot range limiter
    plt.xlim(x_lim_left, x_lim_right)
    plt.ylim(bottom=0)

    # Plot customizations
    plt.xlabel("Time", fontsize=14)
    plt.ylabel("Valid NoRP flux negating quiet sun", fontsize=14)
    plt.title("NoRP quiet sun subtracted flux against time", fontsize=16)
    plt.legend(fontsize=10)
    plt.show()
```

```python


# RSTN plotter - preliminary
def rstn_plotter(
    data_apl_tim,
    data_phf_tim,
    data_apl_flux_peak,
    data_phf_flux_peak,
    data_apl_freq,
    data_phf_freq,
):
    # Plot apl data with loops
    [
        plt.plot(
            data_apl_tim, data_apl_flux_peak[:, i], label=data_apl_freq[i]
        )
        for i in range(data_apl_flux_peak.shape[1])
    ]
    # Plot phf data with loops
    [
        plt.plot(
            data_phf_tim, data_phf_flux_peak[:, i], label=data_phf_freq[i]
        )
        for i in range(data_phf_flux_peak.shape[1])
    ]

    # Ticks declutter
    plt.xticks(data_apl_tim[::2000], rotation=90, fontsize=10)

    # Plot range limiter
    plt.ylim(bottom=0)

    # Plot customizations
    plt.xlabel("Time", fontsize=14)
    plt.ylabel("Valid RSTN flux negating quiet sun", fontsize=14)
    plt.title("RSTN quiet sun subtracted flux again time", fontsize=16)
    plt.legend(fontsize=10)
    plt.show()


# Combined plotter - preliminary
def combined_plotter(
    data_norp_tim_valid,
    data_norp_fi_peak,
    data_norp_peak_time,
    data_apl_tim,
    data_phf_tim,
    data_apl_flux_peak,
    data_phf_flux_peak,
):
    # Plot with loops
    [
        plt.plot(data_norp_tim_valid, data_norp_fi_peak[:, i])
        for i in range(data_norp_fi_peak.shape[1])
    ]
    [
        plt.plot(data_apl_tim, data_apl_flux_peak[:, i])
        for i in range(data_apl_flux_peak.shape[1])
```

```
    ]
    [
        plt.plot(data_phf_tim, data_phf_flux_peak[:, i])
        for i in range(data_phf_flux_peak.shape[1])
    ]

    # Plotting for the peak value
    plt.axvline(x=data_norp_peak_time, color="crimson", linestyle="--")

    # Plot range limiter
    plt.ylim(bottom=0)

    # Plot customizations
    plt.xlabel("Time", fontsize=14)
    plt.ylabel("Valid flux negating quiet sun", fontsize=14)
    plt.title("Quiet sun subtracted flux again time", fontsize=16)
    plt.show()


# %% Sepearate combined fit plotter
# Define fit plotter function
def fit_plotter(arg):
    # Local variable repo
    (
        data_norp_freq,
        data_apl_freq,
        data_phf_freq,
        data_peak_flux_norp,
        data_peak_flux_apl,
        data_peak_flux_phf,
        norp_gyro_param,
        norp_plas_param,
        apl_gyro_param,
        apl_plas_param,
        phf_gyro_param,
        phf_plas_param,
    ) = [arg[i] for i in range(len(arg))]

    # Plot generation
    plt.plot(
        data_norp_freq,
        data_peak_flux_norp,
    )
    plt.plot(
        data_apl_freq,
        data_peak_flux_apl,
    )
    plt.plot(
        data_phf_freq,
        data_peak_flux_phf,
    )
    plt.plot(
        data_norp_freq,
        gyro_model(data_norp_freq, *norp_gyro_param),
    )
    plt.plot(
        data_norp_freq,
        plas_model(data_norp_freq, *norp_plas_param),
```

```
    )
    plt.plot(
        data_apl_freq,
        gyro_model(data_apl_freq, *apl_gyro_param),
    )
    plt.plot(
        data_apl_freq,
        plas_model(data_apl_freq, *apl_plas_param),
    )
    plt.plot(
        data_phf_freq,
        gyro_model(data_phf_freq, *phf_gyro_param),
    )
    plt.plot(
        data_phf_freq,
        plas_model(data_phf_freq, *phf_plas_param),
    )

    # Plot axis scale definer
    plt.xscale("log")
    plt.yscale("log")

    # Plot customizations
    plt.xlabel("Frequencies at peak time (GHz)", fontsize=14)
    plt.ylabel("Valid quiet sun filtered flux", fontsize=14)
    plt.title("Quiet sun flux at peak time", fontsize=16)
    plt.legend(fontsize=10)
    plt.savefig("./media/figure_fit.png")
    plt.close()
```