UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Escola Superior d'Enginyeries Industrial, Aeroespacial i Audiovisual de Terrassa

Bachelor's Degree in Aerospace Technologies Engineering

Evaluation of pulse detection of the Geostationary Lightning Mapper (GLM)

Bachelor's Thesis

Author

Morán Domínguez, Jaime Francisco

Director

Montanyà Puig, Joan

Co-Director

López Trujillo, Jesús Alberto

Escola Superior d'Enginyeries Industrial, Aeroespacial i Audiovisual de Terrassa (ESEIAAT)

—

Terrassa, September 2021

Contents

Nomenclature IV

List of Figures V

List of Tables VII

List of Listings VIII

Declaration of Honour IX

Abstract X

1 Introduction 2

1.1 Object . . . 2

1.2 Justification . . . 2

1.3 Scope . . . 3

1.3.1 Preliminary study . . . 3

1.3.2 Lightning detection data study . . . 3

1.3.3 Lightning detection performance . . . 4

1.4 Requirements . . . 4

2 Background 7

2.1 GLM . . . 7

2.2 ASIM . . . 9

3 Explanation of the resolution algorithm 12

3.1 General resolution algorithm . . . 12

3.2 Initial data management . . . 13

3.2.1 GLM's data order . . . 13

3.2.2 LINET's data handling . . . 15

3.2.3 MMIA's data handling . . . 17

3.3 GLM and MMIA data extraction and conditioning . . . 17

3.3.1 MMIA data extraction and conditioning 18
3.3.2 GLM data extraction and conditioning 20
3.4 Cross-correlation of signals and peak detections 24
3.4.1 Cross-correlation basic functioning 24
3.4.2 Cross-correlation of GLM and MMIA vectors 25
3.4.3 Detection of peaks in GLM and MMIA signals 29

4 Results 33

4.1 Presentation of final results 33
4.1.1 Delays between GLM and MMIA 33
4.1.2 Peak correspondence 35
4.2 Conclusions 38
4.3 Future Continuation 39
4.4 Budget 41

References 42

Appendices 43

A Code of the `main_TFG.py` script 44

B Code for important functions in order of appearance in `main_TFG.py` 51

B.1 Function `GLM_data_ordering.py` 51
B.2 Function `get_MMIA_dates.py` 52
B.3 Function `check_existance.py` 53
B.4 Function `get_linet_timing.py` 54
B.5 Function `MMIA_data_ordering.py` 58
B.6 Function `extract_MMIA.py` 60
B.7 Function `unify_MMIA_data.py` 61
B.8 Function `condition_MMIA_data.py` 63
B.9 Function `extract_GLM.py` 65
B.10 Function `unify_GLM_data.py` 67
B.11 Function `condition_GLM_data.py` 68

B.12 Function `fit_vector_in_MMIA_timesteps.py` 72
B.13 Function `signal_delay.py` 74
B.14 Function `cross_correlate_GLM_MMIA.py` 76
B.15 Function `get_GLM_MMIA_peaks.py` 79

Nomenclature

CCD	Charge Coupled Device
CHU	Camera Head Unit
FOV	Field Of View
GLM	Geostationary Lightning Mapper
GOES	Geostationary Operational Environmental Satellites
GPST	Global Positioning System Time
ISS	International Space Station
LCFA	Lightning Cluster Filter Algorithm
LMA	Lightning Mapping Array
MMIA	Modular Multispectral Imaging Array
MXGS	Modular X- and Gamma- ray Sensor
NetCDF	Network Common Data Form
NOAA	National Oceanic and Atmospheric Administration
OTD	Optical Transient Detector
PHOT	Photometer
TGF	Terrestrial Gamma-ray Flashes
TLE	Transient Luminous Events
TRMM	Tropical Rainfall Measuring Mission

List of Figures

2.1 Illustrations of GLM instrument aboard the GOES-16 satellite, both from [5] . . . 8
2.2 Combined FOV view from the GOES-R series constellation (75 W, 137 W) superimposed on 10-yr of lightning observations from the NASA Lightning Imaging Sensor on board the Tropical Rainfall Measuring Mission (TRMM/LIS) and Optical Transient Detector (OTD) low earth-orbiting satellites, from [5] . . . 9
2.3 Illustrations of ASIM payload and MMIA ubication . . . 10
3.1 Visual representation of the global resolution algorithm . . . 12
3.2 Example of a *Panoply* capture for GLM data of October 30th, 2019 . 14
3.3 Examples of unaltered (red) and filtered (blue) MMIA 777.4nm photometer detection signals . . . 20
3.4 Fast visualization of GLM's data extraction restriction fonts for a generic snippet, day `matches[i]` position `j` . . . 21
3.5 Pre-integration and integrated snippet time vector VS samples . . . 23
3.6 Example of an integrated GLM signal in MMIA timesteps for June 27th, 2020, snippet 2 . . . 23
3.7 Fast visualization of cross-correlation dynamics for `'full'` mode . . . 25
3.8 Representation of example signals `Vector1` (blue) and `Vector2` (red) before and after being cross-correlated and synchronised . . . 26
3.9 Representation cross correlation of GLM and MMIA signals for June 27th, 2020, snippet 2 . . . 30
3.10 Example of GLM and MMIA detected peaks for June 27th, 2020, snippet 2 . . . 31
4.1 Simple example of delay sign convention . . . 34
4.2 Basic delay statistics for GLM and MMIA signals . . . 35
4.3 Distributions of MMIA delays separated for positive and negative, based on average energy of the whole signal and its standard deviation 36

4.4 Example of GLM and MMIA signals displaying their detected peaks in blue, and with common peaks in yellow 37

List of Tables

1.1 Project Requirements 5
3.1 Example of the structure of a given *.csv* file with LINET data 15
3.2 Basic structure of `events` (`linet_times`) variable 16
3.3 Representation of the `MMIA_raw_data`-type variable structure 19
3.4 Example of an output *.txt* for June 27th, 2020, snippet 2 21
4.1 Main work packages and cost associated 41

List of Listings

3.1 Example of user input for GLM directories . . . 14
3.2 Generation of the Cross-Correlation `x` vector regarding even and odd values of `len(Vector1) + len(Vector2)` . . . 27
3.3 Variation of `delay_samples` value considering different lengths of `Vector1` and `Vector2` . . . 28
3.4 Variation of `delay_samples` value considering samples in between same times in GLM and MMIA time vectors . . . 28
A.1 Full code for the `main_TFG.py` script of the program . . . 44
B.1 Full code for `GLM_data_ordering.py` function . . . 51
B.2 Full code for `get_MMIA_dates.py` function . . . 52
B.3 Full code for `check_existance.py` function . . . 53
B.4 Full code for `get_linet_timing.py` function . . . 54
B.5 Full code for `MMIA_data_ordering.py` function . . . 58
B.6 Full code for `extract_MMIA.py` function . . . 60
B.7 Full code for `unify_MMIA_data.py` function . . . 61
B.8 Full code for `condition_MMIA_data.py` function . . . 63
B.9 Full code for `extract_GLM.py` function . . . 65
B.10 Full code for `unify_GLM_data.py` function . . . 67
B.11 Full code for `condition_GLM_data.py` function . . . 68
B.12 Full code for `fit_vector_in_MMIA_timesteps.py` function . . . 72
B.13 Full code for `signal_delay.py` function . . . 74
B.14 Full code for `cross_correlate_GLM_MMIA.py` function . . . 76
B.15 Full code for `get_GLM_MMIA_peaks.py` function . . . 79

Declaration of Honour

I declare that,

the work in this Degree Thesis is completely my own work,

no part of this Degree Thesis is taken from other people's work without giving them credit,

all references have been clearly cited,

I'm authorised to make use of the research group related information I'm providing in this document.

I understand that an infringement of this declaration leaves me subject to the foreseen disciplinary actions by the *Universitat Politècnica de Catalunya - Barcelona TECH*.

Jaime F. Morán Domínguez		September 28th, 2021
Student's Name	*Signature*	*Date*

Title of the Thesis:

Evaluation of pulse detection of the Geostationary Lightning Mapper (GLM)

Abstract

This study evaluates the sensitivity of the Geostationary Lightning Mapper (GLM) versus the Modular Multispectral Imaging Array (MMIA)'s photometer 3, both operating at the oxygen band at 777.4nm of wavelength. To do so, both GLM's and MMIA's data is extracted from pre-processed data files by a Lightning Cluster Filter Algorithm (LCFA) -data classified by time, geolocalization, detection intensity and with an ID number- to be treated by classification of their signals according to Linet detections in Colombia, signal synchronization and peak comparison.

Resumen

En este estudio se evalúa la sensibilidad del sistema GLM (Geostationary Lightning Mapper) frente a la de los fotómetros de MMIA (Modular Multispectral Imaging Array), en concreto el fotómetro 3 captando la luz en 777.4 nm de longitud de onda. Para ello, tanto los datos de GLM como de MMIA son extraídos de los ficheros con información preprocesada por LCFA (Lightning Cluster Filter Algorithm) -datos clasificados por tiempo, geolocalización, intensidad de la detección y con un número identificativo- para posteriormente ser tratados mediante la clasificación sus señales en detecciones dadas por la red de detección de rayos Linet en Colombia, sincronización de señales y comparación de picos de señal.

Introduction

1 Introduction

1.1 Object

The objective of this study is to develop a program to evaluate the pulse detection sensitivity of the Geostationary Lightning Mapper (GLM) aboard the GOES-16 satellite against the sensitivity of the 777.4nm photometer of the Modular Multispectral Imaging Array (MMIA) on the Columbus Module aboard the International Space Station (ISS) over lightning detections in Colombia.

To achieve that goal, given data from both instruments is broken into data snippets according to Colombian Lightning Location Network (LINET) and compared snippet-to-snippet, cross-correlating their signals and outputting their relative delay as well as their detected and non-detected peaks.

1.2 Justification

As comprehension of atmospheric phenomena increase, more effort is put into the study of their consequences and possible potential or risk. One of those atmospheric phenomena is thunderstorm processes, especially lightning activity.

Understanding of those phenomena allows for improvement in many different areas of knowledge as well as it has many direct practical applications. Lightning activity monitoring given by different systems can allow for, for instance, better comprehension of severe thunderstorm lead times and dynamics, early warning of lightning ground strikes or even provide valuable data for improving numerical weather prediction models, as well as decreasing weather prediction uncertainty and false alarm probabilities. In more direct applications, this knowledge can be applied to improving aero- and nautical routing over oceanic regions where lightning activity information is scarce and work as an assistant to storm radar systems in locations where radar coverage is poor [7].

All that valuable information must be given by a lightning detection system in which to rely on to assure detection accuracy in geolocation, time and magnitude. In this study the GLM instrument is evaluated against a MMIA's photometer, both space-based, to account just how reliable its detections really are and how many of them are ignored by this system. This would allow for better future analysis and results in lightning detection studies as well as for better tuning in future detection systems.

While GLM is an operational-purpose instrument with built-in GPS Time (GPST), MMIA is a high time-resolution and sensitivity scientific instrument with own time. This leads to a GLM time accuracy of $2ms$, while MMIA's time accuracy is to up to $20ms$. With this study, high quality lightning detections from MMIA are cross-correlated with time-accurate GLM detections in order to have ASIM data with a time accuracy of $2ms$, 10 times better than before, reporting high-detailed, time-accurate detection data in a high sample rate of $100kHz$.

1.3 Scope

In order to compare signals from GLM and MMIA using cross-correlation, both different datasets must be treated and driven into a similar structure to allow direct analysis. This section schematizes the key elements to proceed from study on the field to results outputting.

1.3.1 Preliminary study

- Review on previous research on lightning detection.
- Review on GLM and MMIA instruments.
- Review on future lightning detection systems.

1.3.2 Lightning detection data study

- Development of a general resolution algorithm.

- Development of tools for extracting GLM and MMIA data from their given file formats to similar data structures.
- Development of tools for trimming GLM and MMIA data into snippets from LINET data.
- Development of tools for GLM and MMIA data conditioning before cross-correlation.
- Development of tools for rejection of trivial snippets.
- Development of a snippet synchronisation system via cross-correlation.
- Development of signal time delay counter.

1.3.3 Lightning detection performance

- Development of tools for peak detection in GLM and MMIA cross-correlated signals.
- Development of tools for accounting detected and non-detected peaks among those instruments.
- Development of tools for accounting difference in order of magnitude in common detected peaks.

1.4 Requirements

Requirements and restrictions on the development of the architecture of the solution algorithm, the result presentation on this report and the developed code is presented in Table (1.1).

Table 1.1: Project Requirements

Requirement ID	***Description***
REQ-1	Comparisons must be made independently of the different instruments' nature.
REQ-2	Data output must be equivalent among different instruments in order to be able to make a comparison.
REQ-3	The developed program and its functions should be based on free software for compatibility reasons.
REQ-4	Accuracy of results must be inside acceptable margins as to base a scientific study upon them.
REQ-5	Verification plots must be included in the report as to understand all processes and their accuracy.
REQ-6	Result plots must be included in the report as to show the final output of the program.
REQ-7	The program must check a minimum correlation among different instruments' data to better determine a match.
REQ-8	The program must be fully modular in order to make it comprehensive and easy to retouch.
REQ-9	Statistics about GLM and MMIA peak detections must be outputted from the program.
REQ-10	The analysis must accept automatization of routines to study n cases without problems, regardless of the size of the input.
REQ-11	A full documentation of the code should be made in the report.
REQ-12	Code documentation must explain clearly every step of the script.

Development

2 Background

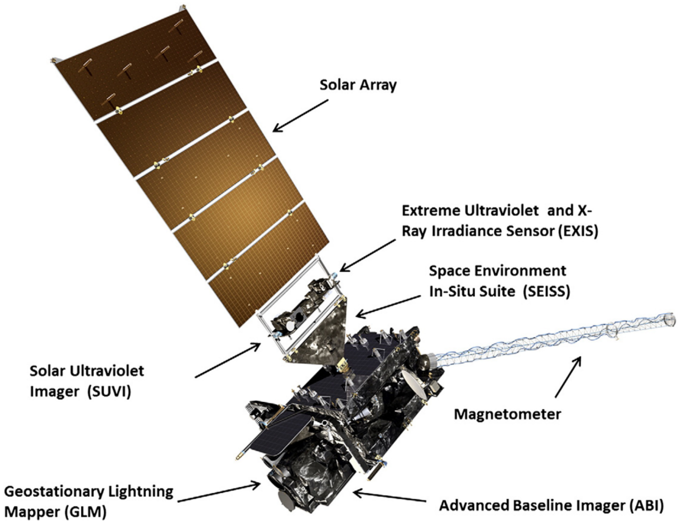
In this section a brief introduction to GLM and MMIA instruments is made, as these are the instruments to be analysed and compared along the study.

2.1 GLM

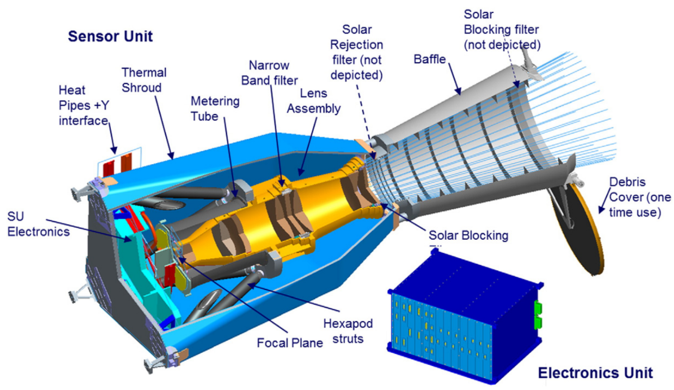
The Geostationary Lighting Mapper (GLM) (Fig.(2.1)) is a high-speed event detector operating at near-infrared wavelength to study both cloud and cloud-to-ground lightning activity 24 hours a day over the American continent. It is a major aid in detecting potentially dangerous storms or weather elements that may affect aviation [5]. The GLM is designed to operate on The Geostationary Operational Environmental Satellite R-series (GOES-R) which becomes part of the original GOES constellation. This set of satellites deployed by the National Aeronautics and Space Administration (NASA) and the National Oceanic and Atmospheric Administration (NOAA) is responsible for providing weather forecasts and warnings. The aim of adding the GLM is to obtain greater accuracy in both timing and prediction.

Currently there are other methods for lightning detection such as the NASA Lightning Imaging Sensor (LIS) and the Optical Transient Detector (OTD) that operate in low Earth orbit. The GLM is able to obtain very similar data and extend its combined climatology to study long-term effects such as climate change for about the next 20 years [4, 1].

To perform well, the GLM has many more requirements than a simple imager. The transient nature of lightning, daylight sampling when there may be solar reflections or spectral characteristics are some of the challenges it faces. A field-of-view (FOV) lens is used together with a narrow-band interference filter and is focused to a high speed Charge Coupled Device (CCD) focal plane. The data is then sent to and processed on the satellite's Local Area Network (LAN). Even though, the LIS also had similar characteristics and made use of the same techniques, with the GLM they have been considerably improved to obtain a much higher accuracy.



(a) Illustration of the GOES-16 geostationary satellite, carrying GLM



(b) Illustration of the GLM

FIGURE 2.1: Illustrations of GLM instrument aboard the GOES-16 satellite, both from [5]

Through the two satellites GOES-E (75 W) and GOES-W (135 W), the focal CCD of the GLM with a resolution of 1372 × 1300 pixels can focus on the storms at any time (see Fig.(2.2)). Being in a geostationary orbit, it has a field of view of practically the whole hemisphere with a resolution at nadir of about 8 km and 14 km for the FOV edge. To achieve such a uniform coverage, the pixel distribution has been densified at the extremes by using smaller pixels that compensate the resolution [3].

The device can detect approximately 86% of the lightning strikes and can even reach 90%. To achieve such good performance it is necessary to use a solar blocking filter at the aperture of the instrument together with a solar rejection filter to limit

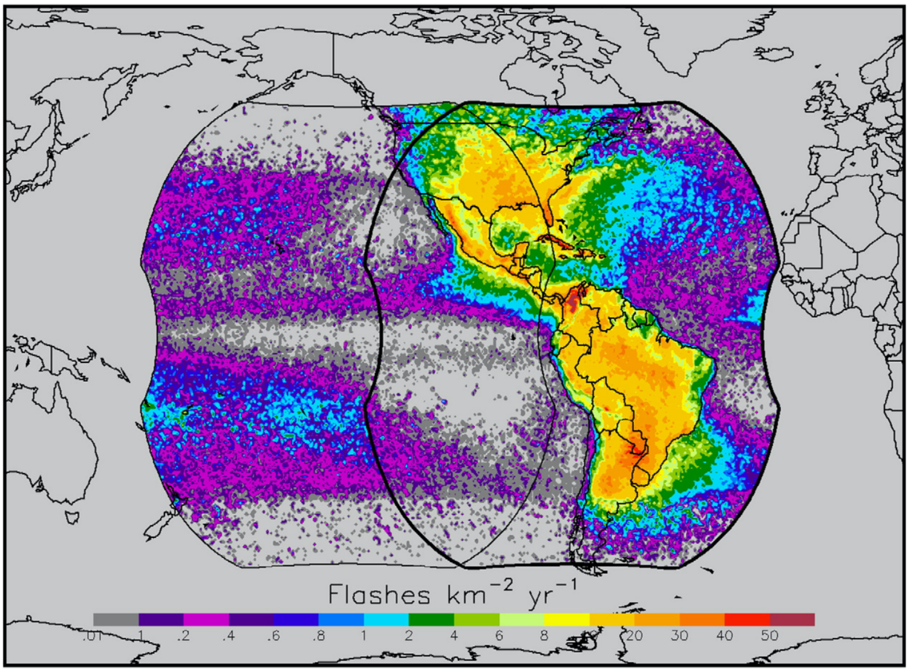


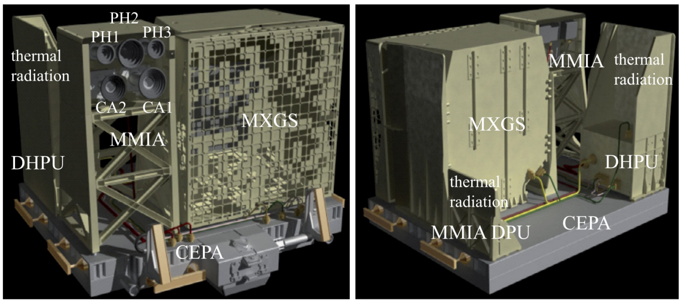
FIGURE 2.2: Combined FOV view from the GOES-R series constellation (75 W, 137 W) superimposed on 10-yr of lightning observations from the NASA Lightning Imaging Sensor on board the Tropical Rainfall Measuring Mission (TRMM/LIS) and Optical Transient Detector (OTD) low earth-orbiting satellites, from [5]

the light outside the band from entering the instrument. In addition, a 1-nm narrow-band interference filter is added to ensure that the 777.4 OI oxygen triplet passes to the detector [5].

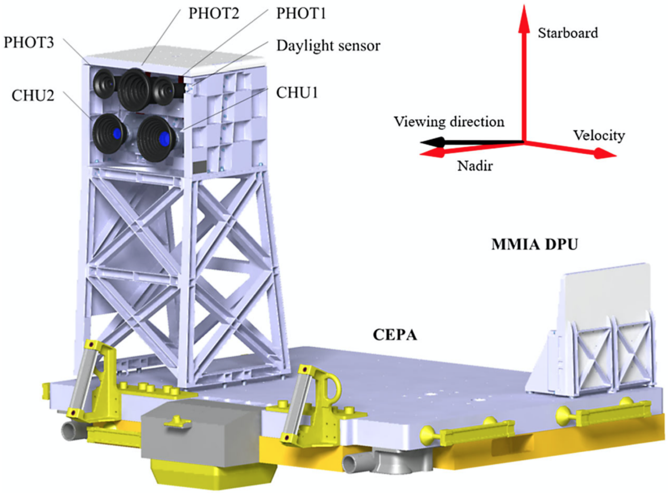
2.2 ASIM

The latest discoveries related with thunderstorms caused a huge amusement in the scientific community. These discoveries show that lightning and thunderstorm can produce electrical breakdown above storms. This event is called Transient Luminous Events (TLEs). TLE can be shown in different forms such as “sprites” (electrical discharges in the mesosphere at 50 to 80 km) and the “blue jets” (streamer type discharges propagating upward from clouds) among others. The electrical band in which these phenomena occur mostly are 337nm, near ultraviolet (NUV).

Due to the great interest of TLEs, a scientific experiment was designed to study this phenomenon, the Atmosphere-Space interactions Monitor (ASIM) (Fig.(2.3)). The ASIM mission’s major scientific objectives are to examine thunderstorm electri-



(a) Illustration of the ASIM payload, from [6]



(b) Illustration of MMIA instrument, from [2]

FIGURE 2.3: Illustrations of ASIM payload and MMIA ubication

cal activity such as lightning, Transient Luminous Emissions (TLEs), and Terrestrial Gamma-ray Flashes (TGFs) by studying the associated emissions in the UV, near-infrared, x-, and gamma-ray spectral bands. The ASIM is formed by two main instruments placed on the Colombus module of the European Space Agency on the International Space Station; the The Modular Multispectral Imaging Array (MMIA) and the Modular X- and Gamma- ray Sensor (MXGS).

The MMIA consists of an array of optical sensors [2]. It is in charge of analysing the TLE phenomenon using three co- aligned photometers and two camaras with the highest sensity, dynamical range and temporal resolution. The instruments are so sensitive to light that they are only operated during nighttime. The two cameras called Camera Head Units (CHUs) are composed by three key elements; an optical assembly consisting of a baffle to reduce stray light and optics hosting a narrow band filter, a focal plane assembly containing an Electron Multiplication Charge Coupled Device (EM-CCD) of high sensitivity, and control and readout electronics capable of reading out up to 12 full frames per second from the sensor. The three co- aligned photometers (PHOTs) are made up of an optical assembly that includes a baffle to limit stray light, lenses that concentrate on the photocathode of a Photo-Multiplier Tube (PMT) in photon counting mode, proximity electronics, and a calibration light emitting diode (LED).

The software is separated into two parts: Boot Software (BSW) and Application Software (ASW), with the BSW running when the computer turns on. Both BSW and ASW are implemented in Ada 2012 using AdaCore GNAT Pro for LEON Bare Board, eliminating the need for an operating system.

MMIA may function in a variety of operational modes focused on main and secondary science goals. The triggered data collection mode enables for the recording of rapid changes in light for the major research objectives of observing transient bright flashes of emissions from thunderstorms. This is used to collect information about lightning strikes, TLEs, meteors, and TGFs. The timed data collection mode enables the performance of programmed periodic observations of set length for secondary research purposes such as auroras.

In conclusion, MMIA is composed of two cameras imaging in the 337 nm and 777.4 nm bands, with a frame rate up to 12 frames per second, and three high-speed photometers in the 180–230 nm, 337 nm and 777.4 nm bands, sampling at rates up to 100 kHz.

3 Explanation of the resolution algorithm

In this section the approach to the problem is explained with detail. Section 3.1 presents a visual representation of the `main_TFG.py` program architecture, sections 3.2 and 3.3 describe the initial approach to data handling and its information extraction for analysis, accordingly. Finally, section 3.4 explains data synchronisation and peak detection on the signals for later result outputting by the program.

3.1 General resolution algorithm

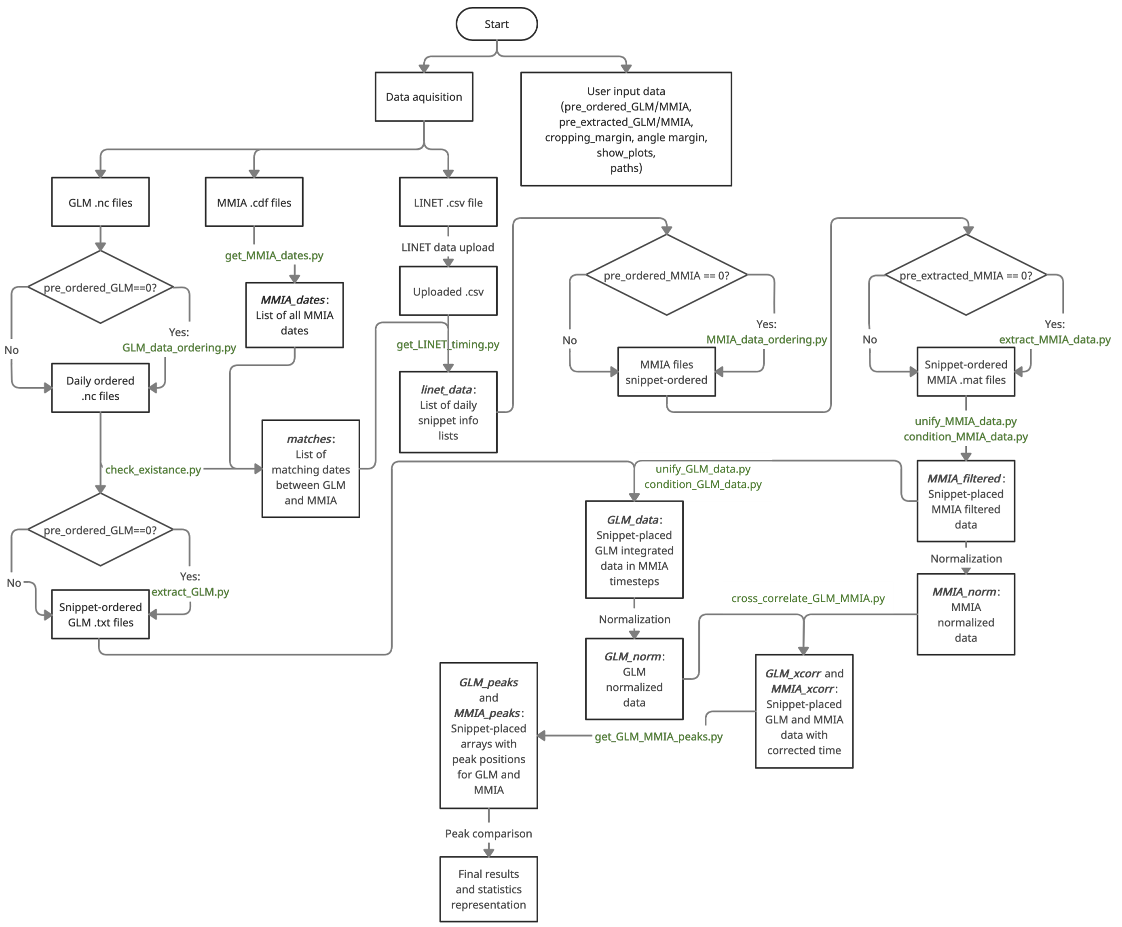


FIGURE 3.1: Visual representation of the global resolution algorithm

3.2 Initial data management

As explained in section 3, this section explains how LINET's, GLM's and MMIA's data is being given as well as how its information is treated into easily manipulable data in order to identify those vector sections with useful information to study.

3.2.1 GLM's data order

GLM data is given by National Oceanic and Atmospheric Administration (NOAA) via a *Google Cloud Platform Repository*, where thousands of *.nc* files are stored with GLM information for 200ms each and ordered by year, day-of-year and hour-of-day. Network Common Data From (NetCDF, giving *.nc* files) is a vastly used multi-platform open-source binary file format to share large amounts of array-oriented data in a way that is self-describing, portable and efficient [11]. In the case of GLM data, those *.nc* files contain information of events, flashes and groups as well as positioning and configuration of the satellite at every moment. Fig.(3.2) gives an idea of the structure of the information inside a GLM *.nc* file and its contents, using the specialized program *Panoply*. As it can be seen it hosts many different variables (rows) with a short description each in the form of arrays (1D).

Once the files of the dates to analyse have been downloaded from the repository, an algorithm to analyse each GLM date range was developed as follows. The program `main_TFG.py` asks for having them all together inside a single directory. It demands the existence of two more void directories, one for hosting multiple daily directories with files inside, and another to host extracted snippet data. Inside USER INPUT DATA section in `main_TFG.py` (see appendix A) the paths to those directories has to be written as a string[1]. Code snippet 3.1 shows an example of this input.

Function `GLM_data_ordering.py` (see appendix B.1) then sweeps along all *.nc* files inside the directory `GLM_files_path` where they are located creating a new

[1]IMPORTANT: Those paths must NOT contain a final '/' (slash) as it will be added when needed.

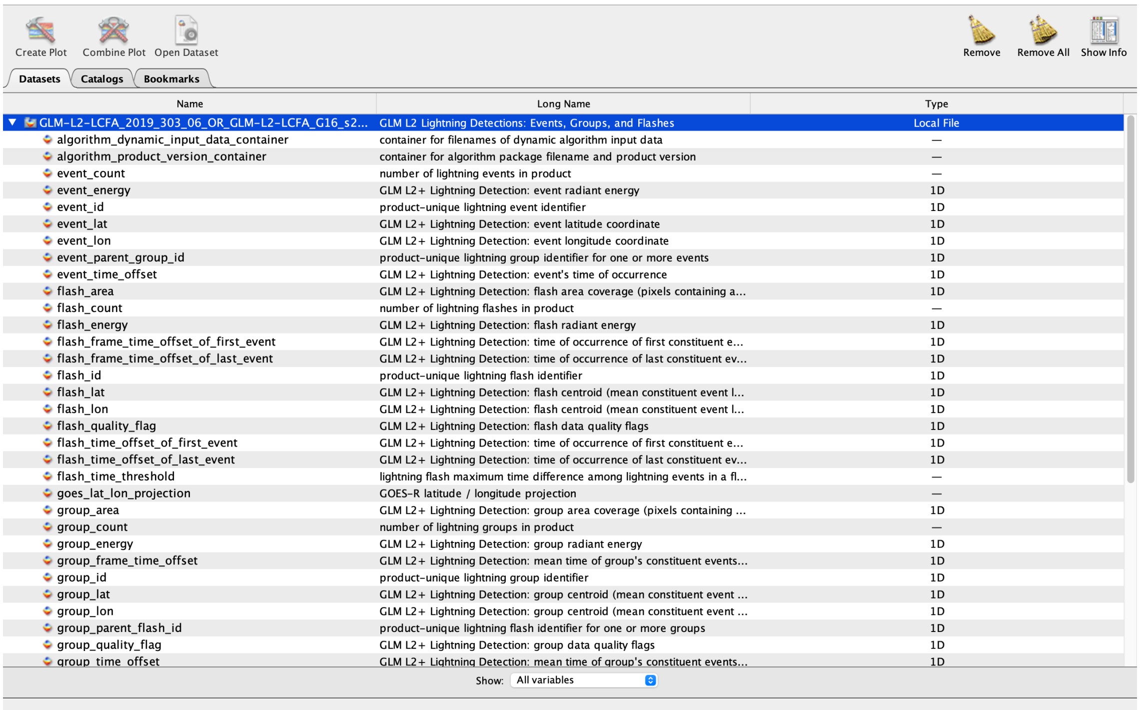


FIGURE 3.2: Example of a *Panoply* capture for GLM data of October 30th, 2019

LISTING 3.1: Example of user input for GLM directories

```python
# Path where GLM's .nc files are located
GLM_files_path =
↪ '/Users/jaimemorandominguez/Desktop/Final/GLM_archivos/nc'

# Path where you want your daily ordered GLM's .nc files to be
↪ located
GLM_ordered_dir = '/Users/jaimemorandominguez/Desktop/Final/GLM_arc
↪ hivos/Dairy_dir'

# Path where you want your daily ordered extracted GLM's .txt
↪ files to be located
GLM_ordered_outputs = '/Users/jaimemorandominguez/Desktop/Final/GLM
↪ _archivos/GLM_output'
```

directory inside GLM_ordered_dir for every different date with existing GLM data, and moving all files with data of that day inside. This new directory is named after the date it contains files of, in the form YearMonthDay.

3.2.2 LINET's data handling

In this study, lightning location data from LINET are used as the ground-truth, and allowed to find the corresponding GLM and MMIA space-based detection as is discussed below. Tab.(3.1) shows the basic structure of the *.csv* file where LINET data is stored in a graphical way for an example of 5 rows.

TABLE 3.1: Example of the structure of a given *.csv* file with LINET data

date_trunc	***ms_linet***	***ka***	***lat***	***lon***	***type***	***id***	***group_id***
2020-02-22 08:37:46	46.294	10.3	3.4295	-75.1912	Lightning	6005	246325
2020-02-22 08:37:46	46.294	10.3	3.4295	-75.1912	Lightning	5990	246325
2020-05-12 23:58:16	16.583	12.6	5.2109	-71.8171	Lightning	5751	313357
2020-05-12 23:58:16	16.583	12.6	5.2109	-71.8171	Lightning	5752	313357
2020-02-23 07:48:37	37.816	16.9	0.4223	-74.8342	Lightning	6957	245729
2020-06-06 02:06:25	25.798	11.3	5.1966	-72.5463	Lightning	25277	0
2020-06-06 02:06:54	53.235	39.4	10.2725	-79.1761	Lightning	25713	0
...	...	...	...	...	...	...	...

As it can be seen LINET data comes with important time (columns `date_trunc` and `ms_linet`) and location (columns `lat` and `lon`) information of detections and their corresponding ID for MMIA's *.cdf* files (column `id`). Ideally, rows are ordered with increasing date and increasing time, i.e. from January 1st at 00:00:00.000 to December 31st at 23:59:59.999, where every row corresponds to an existing LCFA-processed MMIA trigger and belongs into a group marked by its `group_ID` column. Group ID's can be shown, of course, in one or more rows, depending on time and location of its row trigger.

Once all GLM files have been ordered into different daily directories as explained in the previous section, function `get_MMIA_dates.py` (see appendix B.2) returns a list of dates with existing MMIA data as strings in the form YearMonthDay. Function `check_existance.py` (appendix B.3) returns the list `matches` with those dates with existing GLM and MMIA data (it already accounts for different number of dates for each instrument, only returning those that match).

With those date matches computed and with LINET's data already uploaded as a matrix, function `get_linet_timing.py` first gets those lines of the LINET

.csv that correspond to a date inside `matches`. Then separates those lines with an existing value for Group ID from those with a value of 0 (see Tab.(3.1) as an example). For those lines with an existing value a little subset is created storing in a list the starting time and end time (lowest and highest time value of the rows with same Group ID, respectively), minumum and maximum latitude and longitude (following the same philosophy as with time) and a list of all MMIA trigger ID's with that Group ID. In case multiple lines had the same Group ID incorrectly, the program creates new Group ID's for those lines that do not match that group[2]. This list of data defines a *snippet*. Its structure can be seen as:

[**start_time**, **end_time**, **min_lat**, **max_lat**, **min_lon**, **max_lon**, [$\mathbf{ID}_1$, $\mathbf{ID}_2$,...]]

A snippet is a fraction of a signal corresponding to one single group, which is the minimum set of data to compare between GLM and MMIA. All those snippets are stored into a variable called `events`[3], whose structure is widely used along the program. Tab.(3.2) shows a graphical visualization of the structure of this variable. As

TABLE 3.2: Basic structure of `events` (`linet_times`) variable

`matches[0]`	`matches[1]`	`matches[2]`	...	`matches[-1]`
snippet 0	snippet 0	snippet 0		snippet 0
snippet 1	snippet 1	snippet 1		snippet 1
snippet 2	snippet 2	snippet 2		snippet 2
...	...	...		...
snippet -1	snippet -1	snippet -1		snippet -1

seen, `events` variable is just a list of daily lists of snippets. Every snippet information list is stored inside the day position in `matches` of its information (columns), and its position inside the day column is the snippet identification number. Of course every different day will have a different number of snippets according to available detection data for that day. This structure of storing snippets is maintained all over the program, as seen in upcoming sections.

[2]In the case there were highly separated lines with the same Group ID, the program checks if those lines have a maximum separation of 20 lines. If not, new Group ID's are created for those last lines in order not to generate future problems.

[3]Note that in `events` is the name of the variable *inside* the function `get_linet_timing.py`. In `main_TFG.py` this variable is called `linet_times`.

3.2.3 MMIA's data handling

During the development of this study, all MMIA data has been given due to the need of credentials for their download from the server. This data comes in *.cdf* files and, similarly to GLM case, each file is defined by its detection date and hour, and more importantly, its trigger ID number. As in GLM case, the program asks for a directory with all *.cdf* files inside, a void directory for creating new directories and another void directory to store extracted data files. In section USER INPUT DATA of `main_TFG.py` (appendix A) paths to those directories are asked in a similar way as in code snippet 3.1 in section 3.2.1.

Once all snippets have been identified and delimited using LINET's data, MMIA *.cdf* files are ordered in different directories according to the snippet where their ID number appears (see snippet structure in section 3.2.2 and function `MMIA_data_ordering.py` in appendix B.5). This new snippet directory is named after the day of the detections in the form YearMonthDay and the position of the snippet they contain (0 to *n-1*), while copied MMIA's *.cdf* files' names are changed to their ID number. It is important to note that, as MMIA data is ordered following LINET's information just for matches, only those MMIA *.cdf* files regarding a matching date with GLM will be ordered (and extracted, processed and compared), acting as a first filter for MMIA data.

As this GLM and MMIA ordering process is done just once per dataset and all new directories are already installed, boolean variables called `pre_ordered_GLM` and `pre_ordered_MMIA` let bypass this process once it has been done, accelerating the program and avoiding a new unnecessary ordering.

3.3 GLM and MMIA data extraction and conditioning

This section explains how ordered MMIA's and GLM's data is extracted from given files into easily-working arrays and how those arrays are treated and conditioned to be cross-correlated to one another. It is important to note that both functions for explicitly extracting data from the GLM's *.nc* and MMIA's *.cdf* files were given, and no major changes were made to them.

3.3.1 MMIA data extraction and conditioning

Having all MMIA's *.cdf* files ordered by snippet in different directories, the program proceeds to extract their data using a given *MatLab* script that concatenates all *.cdf* files inside a directory and saves their important information as a variable. Function `extract_MMIA.py` (see appendix B.6) hovers over all MMIA's snippet directories calling a *MatLab* engine for every snippet, executing the given *.m* script and returning only a *.mat* file per directory with just time and 777.4 photometer data vectors. This file is called after the directory of the snippet (i.e. *'YearMonthDay_index.mat'*, being index the position of the snippet in that day column) and stored in the outputs directory of MMIA. Some snippets do not output a *.mat* file due to false triggers in the instrument for photometers usually caused by triggers in MMIA's 'CHU' cameras. When a camera starts a trigger using the first frame for storing its data, a decompensation in photometer's signal and time vectors occur causing an error while saving photometer data, so if the case was presented this step is simply avoided and no *.mat* file is outputted. The program prints the day and snippet index for those snippets with no data while writing the variable `MMIA_raw_data`, which in that position remains as type `None`.

A new function, `unify_MMIA_data.py` (appendix B.7) reads all those snippet data tables and writes them inside a similar structure as seen in Tab.(3.2) for LINET's data variable `linet_times`. Tab.(3.3) shows how this data vectors are stored inside `MMIA_raw_data`. As seen, the variable is a list of daily lists of snippet vectors. For every daily position of the variable (i.e. `MMIA_raw_data[0]`, `MMIA_raw_data[1]`, ..., `MMIA_raw_data[-1]`) a list of snippets is stored. The order of daily positioning is the same as in `linet_times` variable, following `matches` order, and the order of snippets is also the same. Snippet information is then given by `MMIA_raw_data[i][j]`, and is presented as a $n\times 2$ matrix of points where `MMIA_raw_data[i][j][:,0]` represents the time vector of points of the snippet and

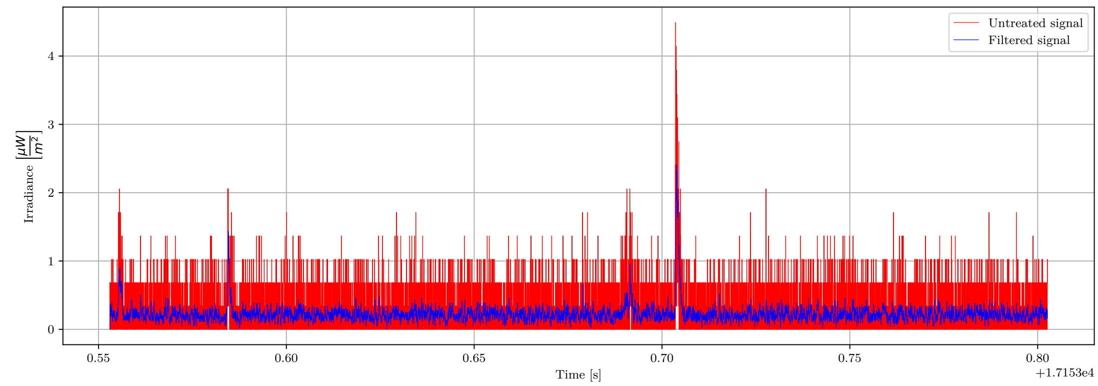
`MMIA_raw_data[i][j][:,1]` represents its signal vector. All this stored data coming

TABLE 3.3: Representation of the `MMIA_raw_data`-type variable structure

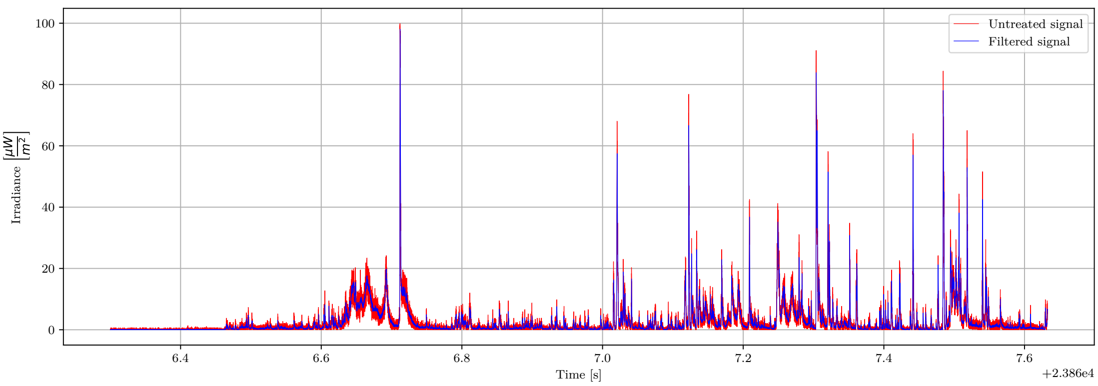
Snippet day position	`matches[0]`	`matches[1]`	...	`matches[-1]`
0	[time[0], signal[0] time[1], signal[1] ... time[-1], signal[-1]]	[time[0], signal[0] time[1], signal[1] ... time[-1], signal[-1]]		[time[0], signal[0] time[1], signal[1] ... time[-1], signal[-1]]
1	[time[0], signal[0] time[1], signal[1] ... time[-1], signal[-1]]	[time[0], signal[0] time[1], signal[1] ... time[-1], signal[-1]]		[time[0], signal[0] time[1], signal[1] ... time[-1], signal[-1]]
...	...	...		...
-1	[time[0], signal[0] time[1], signal[1] ... time[-1], signal[-1]]	[time[0], signal[0] time[1], signal[1] ... time[-1], signal[-1]]		[time[0], signal[0] time[1], signal[1] ... time[-1], signal[-1]]

from the *.cdf* files need to be treated in order to improve the signal quality and the correlation with GLM signal. Function `condition_MMIA_data.py` (appendix B.8) applies a filter to every snippet signal vector to reduce noise, as well as applies an absolute threshold of 1.75 $\frac{\mu W}{m^2}$ to bypass those snippets whose signal vector does not contain any important detection data, just noise. This noise snippets can be caused by triggers in other photometers or cameras of MMIA, while 777.4nm photometer does not detect any significant peak. All snippets are then input to function `fit_vector_in_MMIA_timesteps.py` to assure all time and signal vectors have a sample every exactly 0.00001s, as sometimes MMIA data comes with little time jumps or missing timesteps. Those unexisting samples are filled by a linear regression taking the last and next existing samples as reference points. New time and filtered signal pair of vectors per snippet are stored inside the variable `MMIA_filtered`, which has the exactly same structure as the previous `MMIA_raw_data` structure shown in Tab.(3.3).

Fig.(3.3) shows two examples of unaltered signal vectors as extracted from the *.cdf* files and stored in `MMIA_raw_data` as well as the filtered signals. It is notorious how noise in signal is reduced and how important peaks are more easily detectable, easily seen in Fig.(3.3a). Of course, as signal detection values increase the noise



(a) November 5th, 2020, snippet 1



(b) June 26th, 2020, snippet 2

FIGURE 3.3: Examples of unaltered (red) and filtered (blue) MMIA 777.4nm photometer detection signals

contribution to the final vector is reduced, translating into a clearer curve. It is also important to note how peak values are also altered by their own prominence with respect to the noisy signal, as the filter crops part of their absolute magnitude. Comparing Fig.(3.3a) with Fig.(3.3b) it can be seen how peak magnitude is better conserved in those snippets with less noise (i.e. higher detection energies) and more prominent peaks, where the filtered signal fits the unaltered vector much better.

3.3.2 GLM data extraction and conditioning

With all GLM's *.nc* files daily ordered in separate directories as explained in section 3.2.1, a similar extraction process as followed with MMIA's *.cdf* files is computed with GLM data. Function `extract_GLM.py` (appendix B.9) uses a given *Python* script that concatenates all *.nc* files inside a given directory, extracting all usefull

TABLE 3.4: Example of an output *.txt* for June 27th, 2020, snippet 2

Second of Day	*Lat. Event*	*Lon. Event*	*ID*	*Lat. Flash*	*Lon. Flash*	*Radiance* [J]
23866.493860	3.527283	-71.354897	53555	3.701137	-71.503380	1.40757e-15
23866.496149	3.527283	-71.354897	53555	3.701137	-71.503380	1.78805e-15
23866.498056	3.527283	-71.354897	53555	3.701137	-71.503380	1.38854e-15
...	...	...	...	...	...	...
23867.720686	3.600410	-71.137550	53578	3.530246	-71.177681	1.80707e-15

data from those files that fit inside some restrictions and transcripting it into a .txt output for every date directory. The structure of this *.txt* file can be seen in Tab.(3.4) with an example. As every daily directory contains all GLM's *.nc* files for that day, restrictions in data extraction into the *.txt* file stand for a better delimitation in both time and space for every snippet. Data for those snippet restrictions is given by MMIA variable `MMIA_filtered` as well as by LINET's data variable `linet_times`. For doing so, `main_TFG.py` program hovers over every snippet inside

```
linet_times[i][j] = [start_time, end_time, min_lat, max_lat,
                     min_lon, max_lon, [ID0, ID1, ...  , ID-1]]
```

```
                        [ time[0], signal[0]
                          time[1], signal[1]
MMIA_filtered[i][j] =       ...       ...
                         time[-1], signal[-1] ]
```

Restrictions = `[min_lat-angle_margin, max_lat+angle_margin,`
`min_lon-angle_margin, max_lon+angle_margin,`
`time[0]-cropping_margin,time[-1]+cropping_margin]`

↓

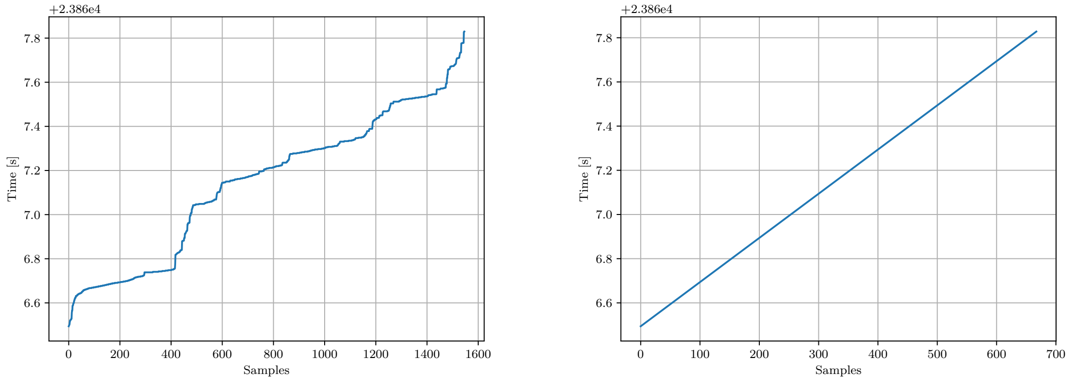
`matches[i]_j.txt`

FIGURE 3.4: Fast visualization of GLM's data extraction restriction fonts for a generic snippet, day `matches[i]` position `j`

`MMIA_filtered` variable, calling the given data extraction function for every MMIA non-`None`-type snippet position (positions that lacked of a *.mat* file as explained in the previous section). For those positions with data (of type `numpy.ndarray`), restrictions for GLM data extraction are the minimum and maximum latitudes and longitudes given by LINET's data, and the first and last time points of MMIA time vector for that particular snippet. A plus of time `cropping_margin` (default 0.2s) and angle `angle_margin` (default 0.5^o) are given to increase probability of matches due to MMIA time uncertainty. Fig.(3.4) summarizes this data acces for a generic snippet. Note again that as `linet_times` and `MMIA_filtered` variables follow the same structure (day order after `matches` and same snippet order inside each day list) every day position - snippet position combination `[i][j]` refers to the same snippet. As in the case of MMIA's *.cdf* files, every GLM snippet *.txt* file is named afer its day and snippet index inside that day (i.e. *'YearMonthDay_index.txt'*), as well as the ordering and data extraction processes can be bypassed once done by setting boolean variables `pre_ordered_GLM` and `pre_extracted_GLM` to 1.

Once GLM data has been extracted, function `unify_GLM_data.py` (appendix B.10) reads all snippet's *.txt*'s, sorts lines by ascending time and writes their information inside the respective snippet position in variable `GLM_raw_data`. This variable has exactly the same structure as `MMIA_raw_data`, seen in Tab.(3.3), but with more information per snippet (all the information from the *.txt*).

Having all GLM data uploaded to the program, its conditioning needs to be done. For every snippet the only important data is the time and radiance vectors, as these are the ones to compare with MMIA. Function `condition_GLM_data.py` (appendix B.11) first checks if the resulting *.txt* had information in it or if the information inside is too poor as to generate a vector (low number of different timesteps) by checking the snippet position in `GLM_raw_data`. If the information contained for that snippet is enough as to generate a vector, it integrates every 0.002s. This step allows for a smoother more reliable curve as well as creates a continuous time vector. Fig.(3.5) shows how the original time vector lacks of some timesteps, while the integrated dataset has a perfect time vector with a sample every 0.002s. The integrated snippet is stored inside variable `GLM_int_data` just before calling function `fit_vector_in_MMIA_timesteps.py` (see appendix B.12). As one of the functions of the program is to cross-correlate GLM and MMIA signals, and cross-correlation works on samples, this is an important step as `fit_vector_in_MMIA_timesteps.py` function expands the GLM snippet into a set of longer time and signal vectors in MMIA timesteps of 0.00001s. To do so, it places the existing GLM timesteps into



(a) Pre-integration (b) Integrated

Figure 3.5: Pre-integration and integrated snippet time vector VS samples

the longer vectors and fills the new blank timesteps with a linear regression between the last and next timestep with data from the previous shorter vector. Of course, this function just adds resolution to the vector without changing its contents or varying the lineality in time. The resulting time and signal vectors are stored into variable `GLM_data`. Both variables `GLM_int_data` and `GLM_int_data` have the exact same structure as shown for `MMIA_raw_data` in Tab.(3.3). Fig.(3.6) shows an example of an integrated and expanded GLM signal.

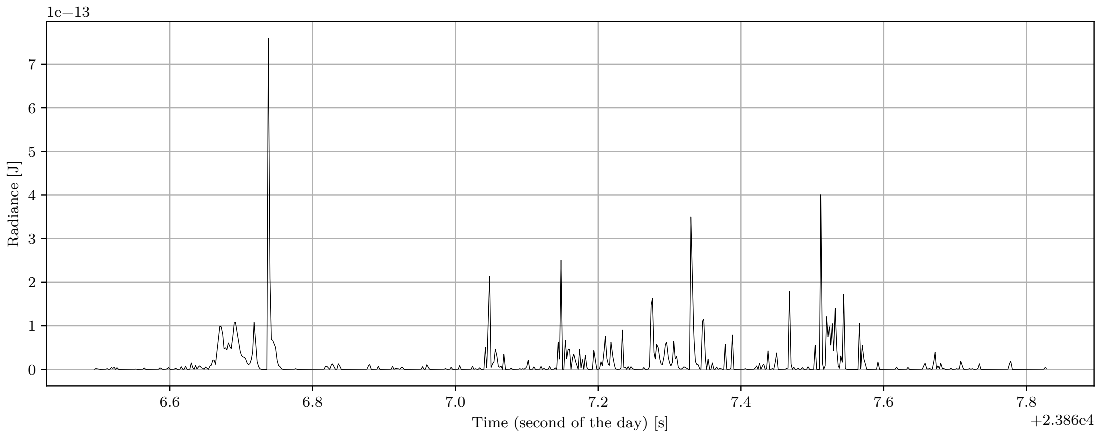


Figure 3.6: Example of an integrated GLM signal in MMIA timesteps for June 27th, 2020, snippet 2

3.4 Cross-correlation of signals and peak detections

In this section it is explained how basic cross-correlation works and how it is implemented to correlate GLM and MMIA signals, as well as how those signals are treated to detect their peaks and then extract some statistics values as is discussed below.

3.4.1 Cross-correlation basic functioning

Cross-correlation is a method for computing the similarity between two signals. In order to do so, two arrays of data `Vector1` and `Vector2` are inputted into the correlation function (in this study `correlate()` function is used from *Python* package `scipy.signal`), which slides `Vector1` over `Vector2` over every overlapping position from `Vector1[0]-Vector2[-1]` to `Vector1[-1]-Vector2[0]` [10]. For every position a correlation factor is computed based on the similarity of the two signals (higher similarity translates into a higher correlation factor), outputting a new vector of correlation factors and length `len(xcorr_factors)` = `len(Vector1)+len(Vector2)-1`[4]. See Fig.(3.7) for a visual respresentation of the process.

Knowing when those signals resemble the most, one can easily move one of them to make it fit perfectly between them. Taking as an example two simple signals of same length (for example, 10) and only one peak of value *1* in a different position for every signal (Fig.(3.8a) plots those signals):

```
Vector1 = [0,0,0,0,0,0,0,1,0,0]
Vector2 = [0,0,0,1,0,0,0,0,0,0]
```

One can see how `Vector1` is delayed by 4 samples with respect to `Vector2`, as it has its peak 4 samples after `Vector2`'s peak. This is translated into a maximum in

[4]The correlation function allows for 3 different modes for cross-correlation. Mode `'full'` is the one used in the study and therefore explained, where the two vectors slide for every overlapping position. Mode `'same'` returns a vector of `len(xcorr_factors)` = `max(len(Vector1), len(Vector2))` and centered with respect to the `'full'` mode, and mode `'valid'` computes cross-correlation just for those overlepping positions where every sample of one vector overlaps a sample of the other, returning a vector with length `len(xcorr_factors)` = `max(len(Vector1), len(Vector2))` - `min(len(Vector1), len(Vector2))` + 1 [8].

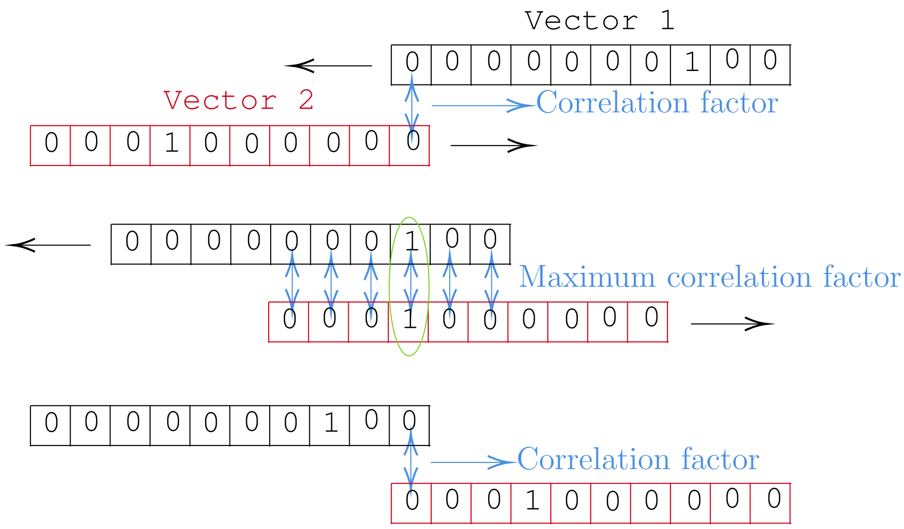


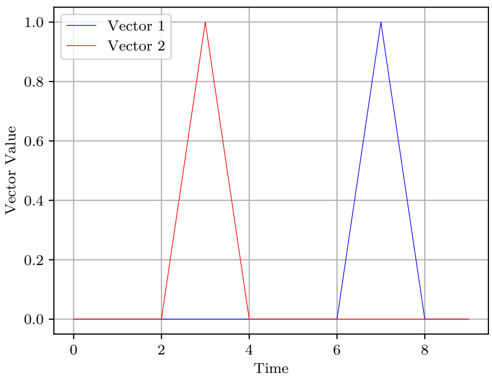
FIGURE 3.7: Fast visualization of cross-correlation dynamics for `'full'` mode

`xcorr_factors`'s position where the two positions with 1 in `Vector1` and `Vector2` match, as the correlation factor in that position increases. Assigning a value to every `xcorr_factors` position where the center position gets a 0, values on the left of the center position get single-spaced negative numbers and values on the right get single-spaced positive positions, this new vector `x` tells the delay samples of `Vector1` with respect to `Vector1`. Following the previous example, `xcorr_factors` and `x` are presented below. Note how the value inside the `x` vector in position where `xcorr_factors` has its absolute maximum tells the delay. Fig.(3.8c) show both signals overlapped represented over samples, as well as `xcorr_factors` over `x`. Finally, Fig.(3.8b) shows `Vector1` and `Vector2` after cross-correlation after using the delay value to align them properly.

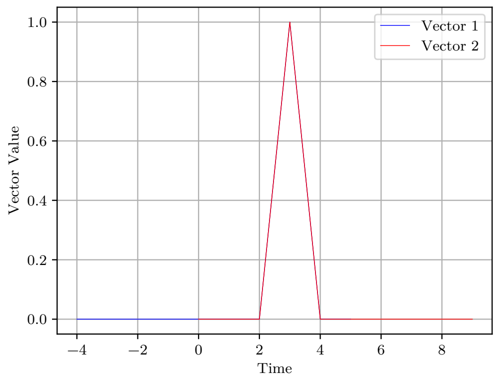
```
xcorr_factors = [0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0]
 x = [-9,-8,-7,-6,-5,-4,-3,-2,-1,0,1,2,3,4,5,6,7,8,9]
```

3.4.2 Cross-correlation of GLM and MMIA vectors

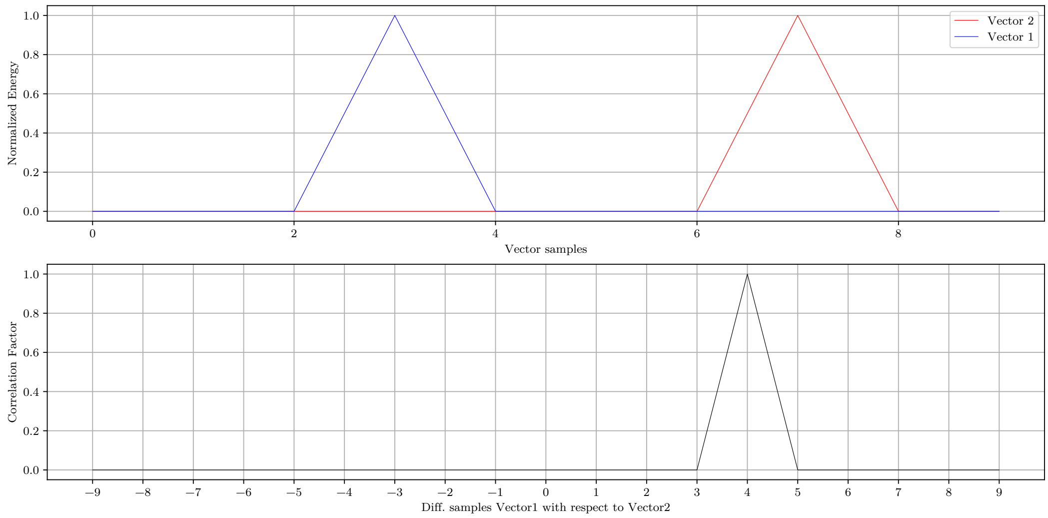
While synchronising GLM and MMIA signals some more steps must be done in order to get an overall good agreement. Although the cross-correlation process follows the same philosophy as in the previous section, the given example considered same-length vectors, an even value of `len(Vector1) + len(Vector2)` and most



(a) Signal representation before XCorr



(b) Synchronised signals after XCorr



(c) Signal representation by samples and correlation factor per overlapping position

FIGURE 3.8: Representation of example signals `Vector1` (blue) and `Vector2` (red) before and after being cross-correlated and synchronised

importantly, both vectors followed the same time vector. While analysing GLM and MMIA's signals, any length of those vectors can be input to the `cross_correlate_GLM_MMIA.py` function (see appendix B.14) and there is no need for GLM signal vector to be in the same time vector as MMIA's.

Firstly, `cross_correlate_GLM_MMIA.py` function calls `signal_delay.py` function (appendix B.13) to compute the real delay of the GLM signal with respect to MMIA's. The first step to compute this delay is exactly the same as in the example before, but considering even and odd values for `len(Vector1) + len(Vector2)`, which would cause problems. Code snippet 3.2 shows how the `x` vector is generated with the consideration in mind, deleting the even or odd problem. In this case, `data1` corresponds to GLM signal vector, while `data2` corresponds to MMIA signal vector. Note that when `len(Vector1) + len(Vector2)` is an even number the

Listing 3.2: Generation of the Cross-Correlation `x` vector regarding even and odd values of `len(Vector1) + len(Vector2)`

```
    len_x = len(data1)+len(data2)-1
    x = np.empty(len_x)

    for i in range(len_x):
        if (len_x % 2) == 0: # Even number
            x[i] = (i - (len_x/2))
        if (len_x % 2) != 0: # Odd number
            x[i] = (i - (len_x/2 - 0.5))
```

resulting `x` vector has an odd length and has the exactly same structure as in the example, with a 0 in the center position. If the length sum is odd, the `x` vector has even length and 0 value falls on the first position of the second half of the vector:

$$x_{\text{odd length sum}} = [0,0,0,0,0,1,0,0,0,0]$$
$$x_{\text{even length sum}} = [0,0,0,0,0,1,0,0,0,0,0]$$

The next step is to account for different lengths of `Vector1` and `Vector2`. Code snippet 3.3 shows how the delay value given by vector `x` in the position where `xcorr_factors` has its maximum value does not match the real delay when the signal vectors have different lengths. Note how a correction of 0.5 samples is made to delay when `len(Vector1) + len(Vector2)` is odd to account for the position in `x`. Once the even/odd length sum and different lengths problems are solved, the given delay represents the number of samples GLM signal is shifted with respect to MMIA signal if both vectors started from the same timestep. As this is not the case (every signal vector has its own time vector) another readjustment is needed. Code snippet 3.4 shows this correction, adjusting `delay_samples` value by getting the difference in samples between same times in GLM and MMIA time vectors.

This value, `real_delay_samples`, is the delay in samples of MMIA signal with

LISTING 3.3: Variation of `delay_samples` value considering different lengths of `Vector1` and `Vector2`

```
    # Delay samples accounting actual positioning due to different lengths:
    max_factor_pos = np.where(xcorr_factors == max(xcorr_factors))[0][0]

    if ((len(data1)+len(data2)) % 2 == 0): # len(x) is Odd
        delay_samples = x[max_factor_pos]+(len(data1)-len(data2))/2

    if ((len(data1)+len(data2)) % 2 != 0): # len(x) is Even
        delay_samples = x[max_factor_pos]+(len(data1)-len(data2))/2 + 0.5
```

LISTING 3.4: Variation of `delay_samples` value considering samples in between same times in GLM and MMIA time vectors

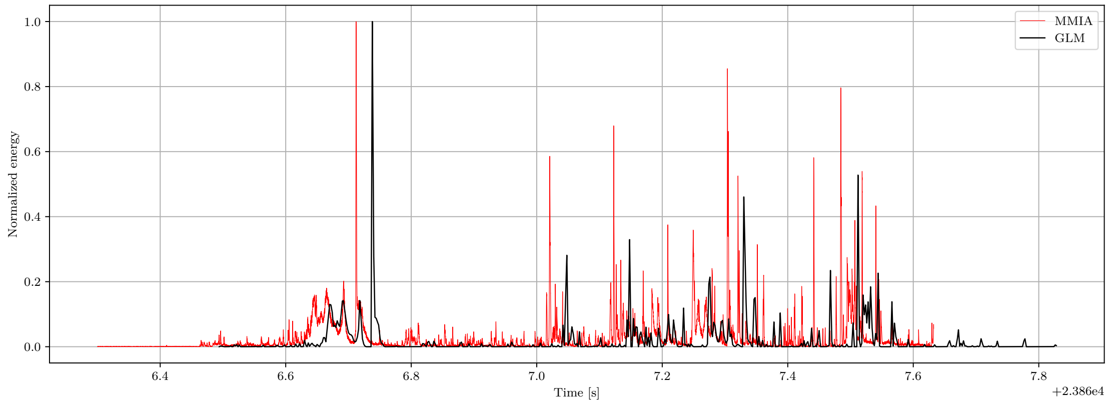
```
    # Delay samples accounting actual positioning due to time:
    if data1[0,0] > data2[0,0]: # GLM vector starts later
        pos_MMIA_start_GLM = np.where(data2[:,0] <= data1[0,0])[0][-1]
        real_delay_samples = delay_samples + pos_MMIA_start_GLM
    elif data1[0,0] < data2[0,0]: # GLM vector starts earlier
        pos_GLM_start_MMIA = np.where(data1[:,0] <= data2[0,0])[0][-1]
        real_delay_samples = delay_samples - pos_GLM_start_MMIA
    else:
        real_delay_samples = delay_samples
```

respect to GLM signal accounting for different lengths of their signal vectors, different even/odd value for their length sum and considering different time vectors for each signal vector. A positive value of this variable means GLM signal is behind MMIA's (MMIA anticipated), while a negative value means GLM signal came before MMIA signal (MMIA delayed). After `signal_delay.py` functions returns this value to `cross_correlate_GLM_MMIA.py`, it just moves the MMIA signal by changing its time vector, adding '`real_delay_samples`' times MMIA's period, 0.00001s. GLM time is taken as reference because of its better time resolution of $2ms$. That is the reason behind always moving MMIA signal instead of GLM's, regardless of the sign of the delay. Delay value per snippet is stored in variable `delays` as it is an important piece of data to know how time-shifted is GLM data with respect to MMIA's.

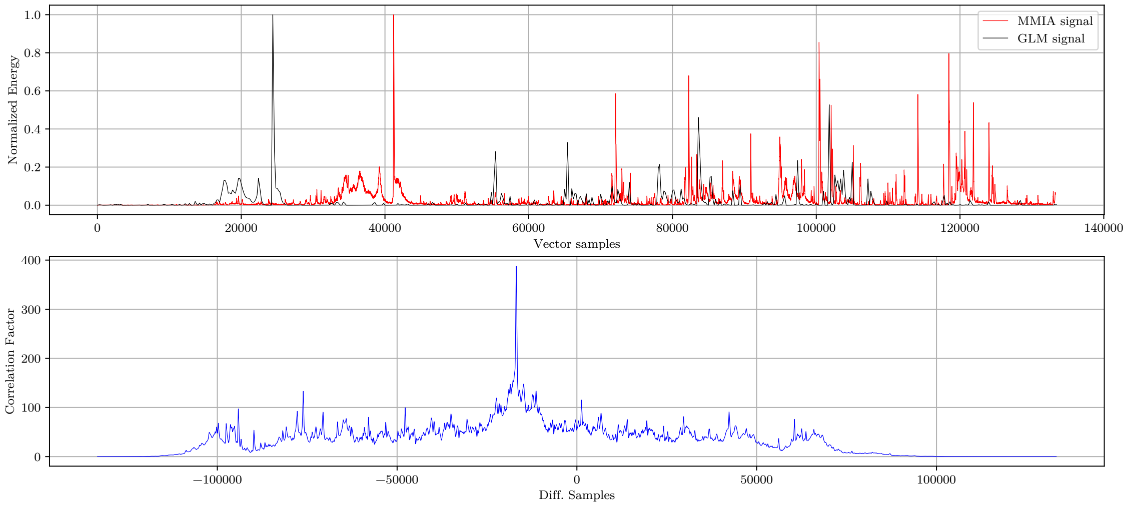
Fig.(3.9) shows an example of uncorrelated signals between GLM and MMIA signals ready to be synchronised. In Fig.(3.9a) both signals can be seen plotted with their respective time vector. Fig.(3.9b) shows those signals plotted by samples and the correlation factor for every position. Note that as those signals are plotted by samples before the delay corrections, time vector is ignored, and the delay seen is not the real one as neither is the difference in samples given by the position of the maximum correlation factor as explained before. Finally, Fig.(3.9c) shows the cross-correlated, synchronised GLM and MMIA signals over their time vectors. For this example, June 27th 2020, snippet 2, the delay is of 2587 samples (positive, MMIA anticipated as seen in the figure) or $0.02587s$ (very close to MMIA maximum time accuracy of $20ms$).

3.4.3 Detection of peaks in GLM and MMIA signals

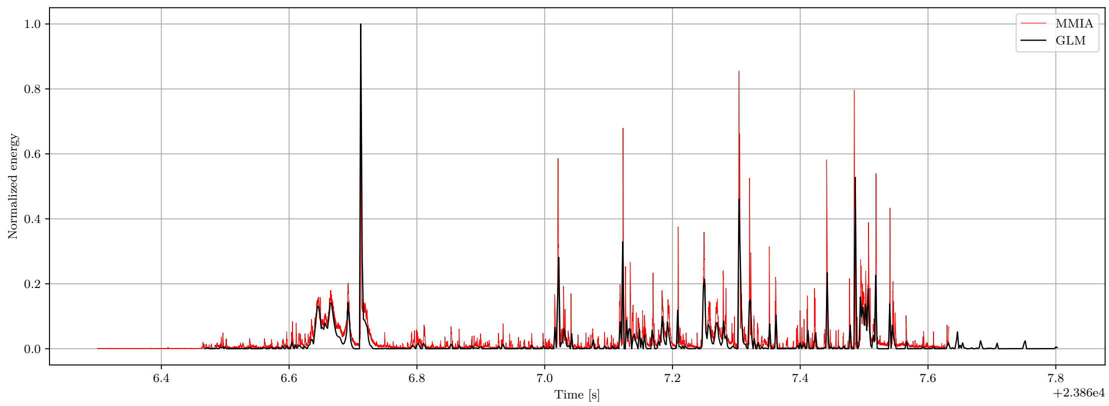
After cross-correlating GLM and MMIA signals both can be directly compared. Function `get_GLM_MMIA_peaks.py` (appendix B.15) first trims correlated GLM and MMIA signals on every snippet to the same length and finds all prominent peaks on both signals, returning a list of indexes where peaks have been detected for every snippet, storing this data in variables `GLM_peaks` and `MMIA_peaks`, both following a similar structure as seen in previous snippet-structured variables (a list of snippet data inside a daily list following `matches` order). Fig.(3.10) shows an example of both GLM and MMIA signals over samples with their detected peaks marked.



(a) Signal representation before XCorr

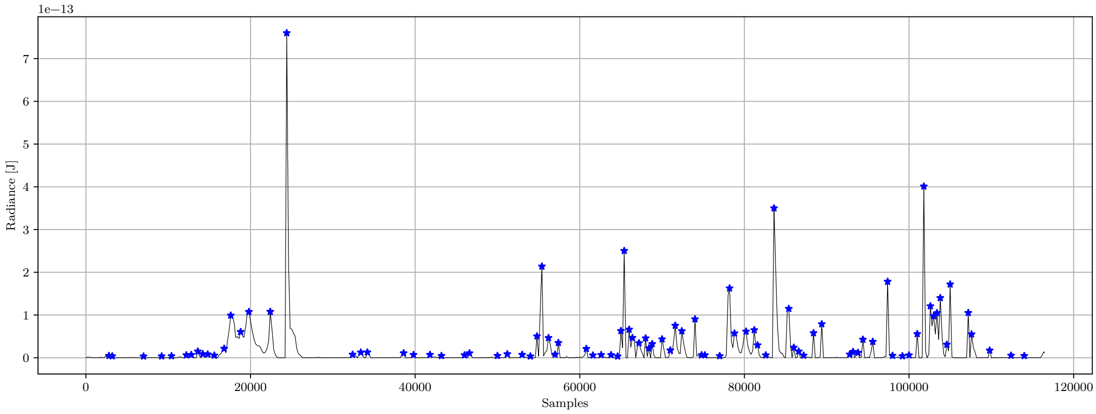


(b) Signal representation by samples and correlation factor per overlapping position

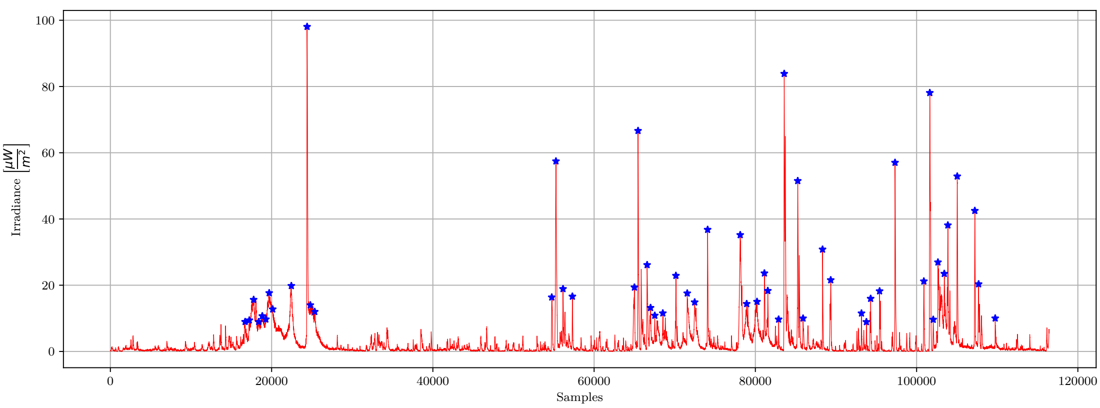


(c) Synchronised signals after XCorr

FIGURE 3.9: Representation cross correlation of GLM and MMIA signals for June 27th, 2020, snippet 2



(a) GLM signal



(b) MMIA signal

FIGURE 3.10: Example of GLM and MMIA detected peaks for June 27th, 2020, snippet 2

Results

4 Results

In this section final results are presented after all data management and treatment. Section 4.1 explains all results from the analysis, section 4.2 presents final conclusions while section 4.3 shows possible avenues for improvement for better and more consistent outputs, and finally, section 4.4 details the total budget for developing this study.

4.1 Presentation of final results

This study covers the detection of lightning for 114 different dates in 2020. For every day, an average of 5.39 snippets have been found. A total of 1507 MMIA triggers were studied, obtaining 615 different snippets, out of which only 340 were suited for cross-correlation and peak comparison (55.28%). Snippets may be discarded if lack of MMIA or GLM data is found after extraction, or if the extracted data is too poor as to conform a snippet. This lack of data may be given by no GLM or MMIA datafiles for a particuar event, or by no detections in the time and space restrictions for a snippet (LINET captured a lightning while MMIA did not have Colombia in its FOV or simply GLM and/or MMIA didn't detect anything, for example). After comparing those snippets that could be analysed, some important statistics can be extracted. In the following sections delays between GLM and MMIA data are studied for a better comprehension of which instrument delays, how often and how much, as well as a better study of signal peaks is explained.

4.1.1 Delays between GLM and MMIA

While cross-correlating signals the delay of MMIA with respect to GLM signal in samples for every snippet was stored in order to compare the results. The total average delay (a positive delay means that MMIA signal anticipated as shown in the simple example of Fig.(4.1)) is $\overline{total\_delay}_{samples} = -1423.72$ samples or $\overline{total\_delay}_{s} = -0.014$ seconds, meaning the average MMIA displacement among

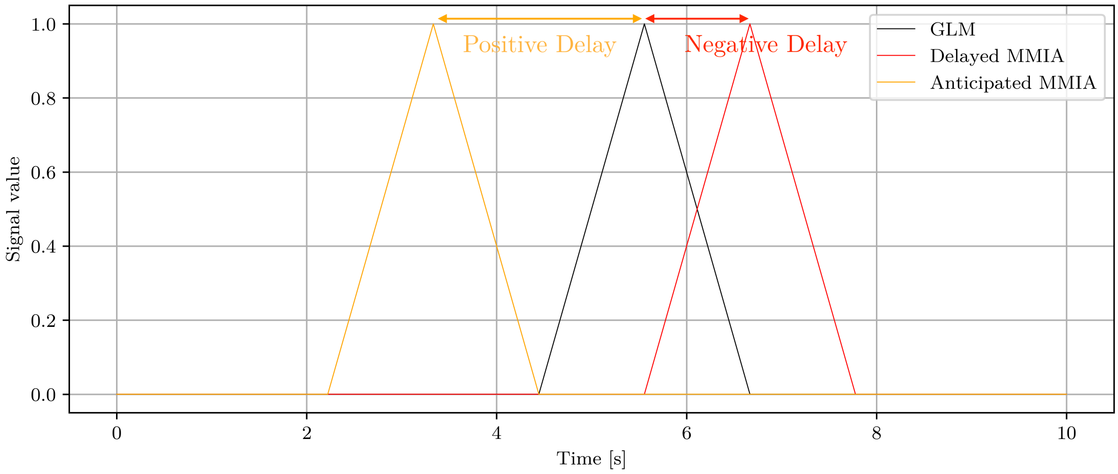
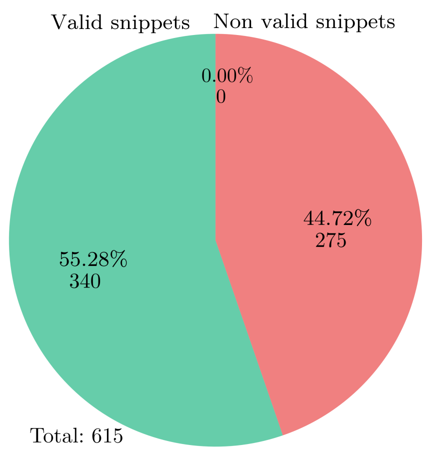
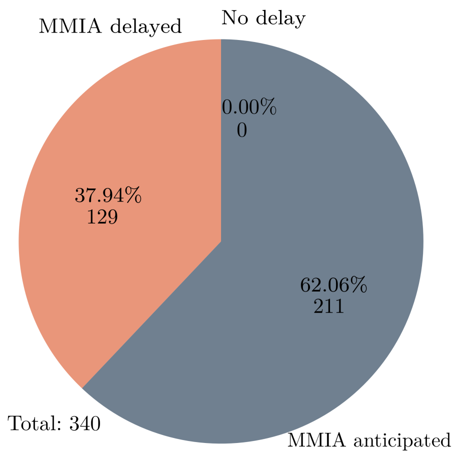


FIGURE 4.1: Simple example of delay sign convention

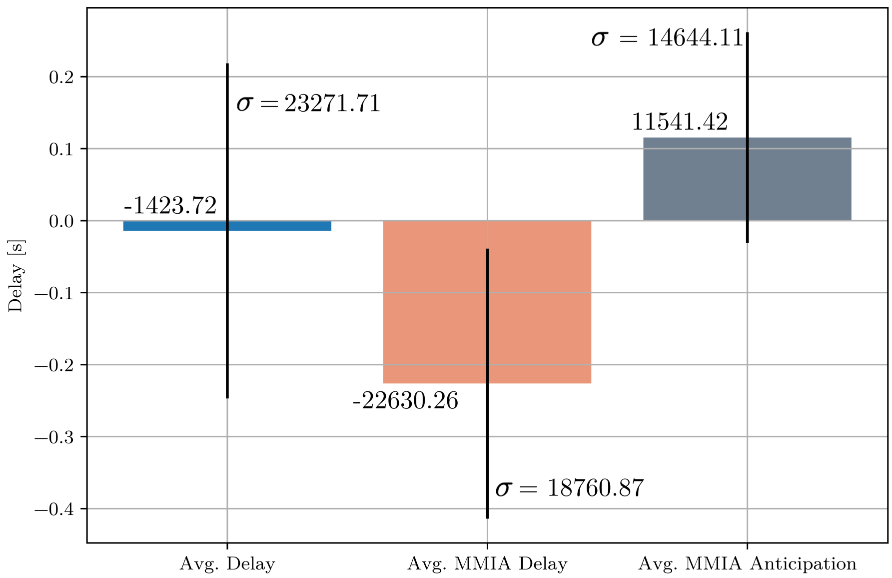
all 340 studied snippets. Although it may seem that MMIA uses to delay more often (negative value), 62.06% of the snippets show it actually anticipates more (211 of GLM over 129 for MMIA). This means that average MMIA delay time is (in absolute value) higher than anticipation time. Indeed, average MMIA anticipation is $\overline{MMIA\_anticipation}_{samples} = 11541.42$ samples (0.115 seconds) while average delay is $\overline{MMIA\_delay}_{samples} = -22630.26$ samples (-0.226 seconds), with standard deviations being $\sigma_{GLM} = 14644.11$ and $\sigma_{MMIA} = 18760.87$ samples (0.146 and 0.188 seconds), respectively. Fig.(4.2) shows this data for easier understanding. It is also interesting to analyse the relationship between the energy of the detections and the probability of being delayed. Fig.(4.3) shows separated scattered data for MMIA-anticipated snippets and MMIA-delayed snippets, where all snippet delays are displayed as a function of the average radiance of the snippet and its standard deviation. As clearly seen, most of the snippets fall into a low-energy, low-delay zone. It can also be seen how as average energy increase, or more peaks are present (higher standard deviation), the delay in the signal decreases. In those cases where MMIA was anticipated a bigger concentration of snippets is found in the low-low zone, but the gradient of delay vs average energy and delay vs average energy is higher than in MMIA's delays (it is easier to have a low delay if the snippet has a high average energy or many peaks on MMIA anticipated-snippets than in MMIA delayed-snippets).



(a) Valid snippets over total snippets



(b) Delayed instrument per snippet



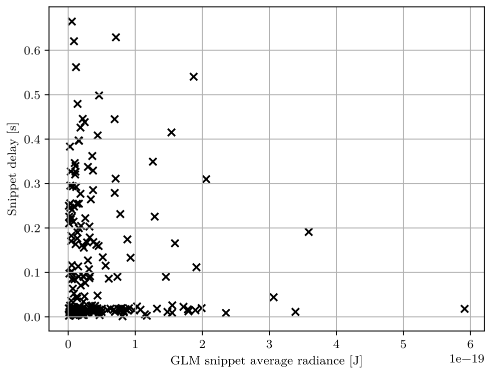
(c) Averages in delays

FIGURE 4.2: Basic delay statistics for GLM and MMIA signals

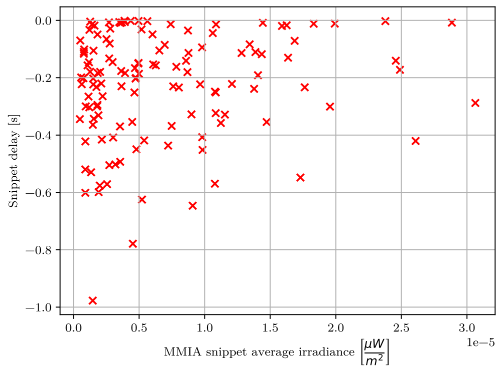
4.1.2 Peak correspondence

Once GLM and MMIA have correlated their signals their peak values and frequency distribution are calculated.

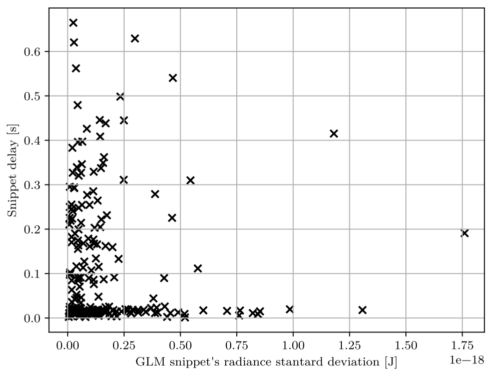
After computing all 114 dates, GLM has, in average, 10.84 peaks per snippet, while MMIA presents an average of 24.29 peaks per snippet (MMIA detects more peaks than GLM of about 44.63%). A function compares both peak vectors in order to determine what peaks on one instrument were detected by the other instrument



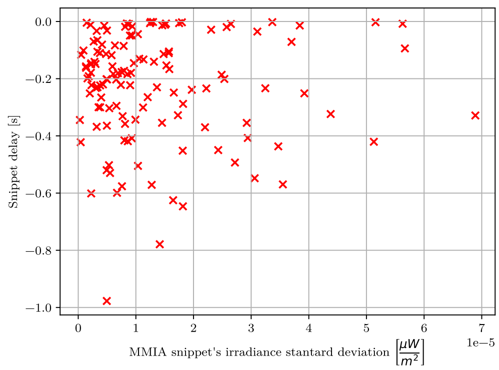
(a) MMIA anticipations VS Average Radiance distribution



(b) MMIA delays VS Average Radiance distribution



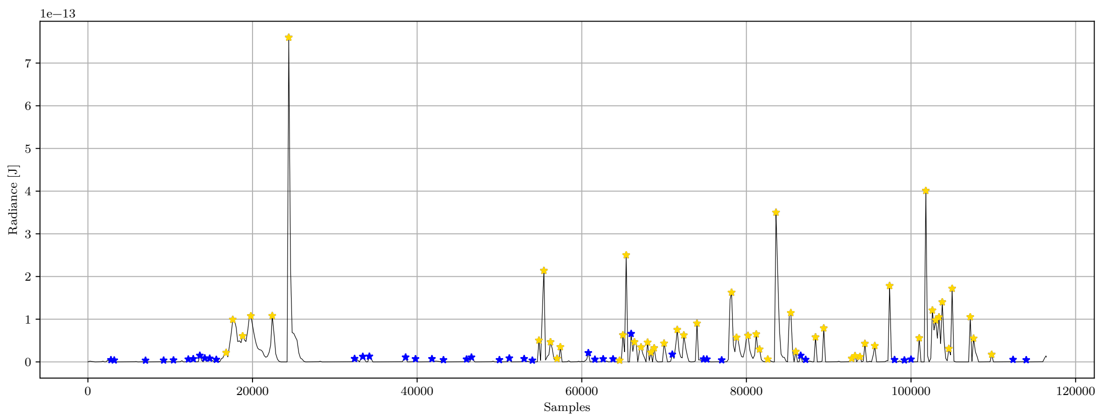
(c) MMIA anticipations VS Std. Deviation distribution



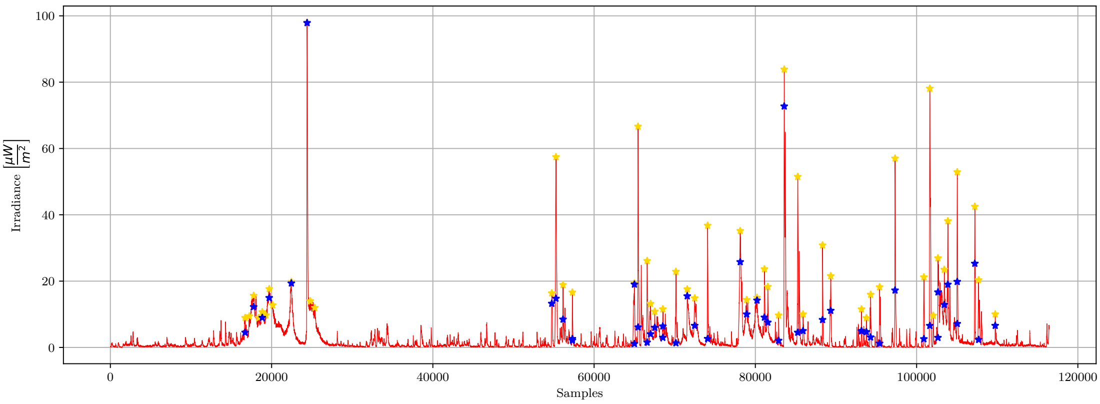
(d) MMIA delays VS Std. Deviation distribution

FIGURE 4.3: Distributions of MMIA delays separated for positive and negative, based on average energy of the whole signal and its standard deviation

and vice versa. On GLM, from every average of 10.84 peaks per snippet, 7.6 peaks are detected by MMIA. That means that 70.42% of GLM maxima are detected by MMIA. On the other way around, numbers flip. For every MMIA average of 24.29 peaks per snippet, only those 7.6 are detected (of course, average detected peaks on both instruments simultaneously does not vary). Exactly, just 36.97% of MMIA's peaks are detected by GLM. Fig.(4.4) shows the example snippet displaying common peaks. The function that compares peaks starts by hovering over all GLM detections and trying to find the nearest MMIA detection in vector position, giving preference to higher peaks.



(a) GLM over samples displaying detected peaks



(b) MMIA over samples displaying detected peaks

FIGURE 4.4: Example of GLM and MMIA signals displaying their detected peaks in blue, and with common peaks in yellow

4.2 Conclusions

To conclude this report, some considerations have to be taken. One of the main purposes of this project was to develop useful tools for determine the detection sensitivity of the Geostationary Lightning Mapper with respect to the Modular Multispectral Imaging Array. Another important step this study has made is to reduce MMIA's uncertainty from about 20ms to 2ms thanks to LINET's and GLM's data, of much more timing precision.

In the development of those tools several difficulties have been found. First of all, the main architecture of the analysis was designed, showing what functions where needed for the whole analysis. Snippets were defined and trimmed to evaluate their single-piece-of-analysis capability.

As seen in previous sections, GLM instrument lacks of sensitivity when compared to MMIA's photometer of much more resolution. When given data contains signals of low energy, it has been proved how MMIA data is more likely to come more delayed than in those high energy signals with prominent peaks. It also has been shown how GLM's low sensitivity concatenates a lower peak detection number than MMIA.

This study also opens an avenue to consolidate a program to easily automatise the process of MMIA normalisation using GLM data and its comparison, as it is a widely frequented process during thunderstorm and lightning activity studies.

4.3 Future Continuation

Although this study has been performed in a period of 7 months and great attention to detail has been put into every function, there are some key points in the way to continue with the study. As the main weight of the process has been the programming process of the resolution, future continuation on this work should follow the same line.

All the process is maintained by the importance of the snippets, as they are the minimum datasets for comparison. As so, their definition is key to a good cross-correlation and posterior analysis, as well as their initial treatment. One of the main points to retouch in the program is MMIA filter's parameters, when the initial non-processed data is smoothered (see section 3.3.1). The optimal filter should straighten the signal as much as possible without losing any shape, specially when finding prominent peaks.

Focusing on snippets, their characterization must be perfect in order to get good results in the following steps as mentioned before. Not all snippets are valid for analysis, as many of them are noise signals in MMIA, for example, or extremely short data vectors in GLM. Although the program detects most of those problematic snippets and takes them out of next steps, a good way of continuing the study would be to better characterize those non-valid snippets and even looking for a better way to understand their procedence. If not deleted as soon as possible, final statistics may be highly biased by those non-constructive snippets.

Another important step for the analysis of the signals is done in cross-correlation. So far the presented function cross-correlates and synchronises any two sets of data regardless of their length or time vectors, as explained in section 3.4.2, successfully. In order to mantain a good cross-correlation, good quality snippets should be used, returning to the main previous idea. If not, extreme values of delays may be seen in further statistics that can lead to great confusion regarding GLM timing.

One of the most improvable points of all the program is executed as peak detection. Due to the noisy MMIA signal, which is not constant for all snippets, it is difficult to detect a real peak from a noise peak. This problem leads to snippets with large amounts of 'ghost' peaks than can alter future statistics reducing drastically GLM detection rate, when those peaks are just fake. So far a threshold is computed as the 90th percentile of all snippet signal vector, assuming most of the samples are located as noise in the lowest energy levels. A retouch on peak detection functions' parameters would probably allow for more accurate peak detections, improving results drastically. It would be even useful to cross-correlate the peak position vectors to compare how many peaks were detected by both instruments even better.

Finally, although some other considerations such as output data quantity, code cleanup and reestructure or optimization for faster code using less computer resources could be done, there is a key point during data ordering processes, explained in section 3.2. All file ordering is done by using the `os` Python library, which allows for direct command line orders using the function `os.system()`. This function takes as a input a command line order as a string, and just executes it on a terminal. As all the study has been developed using a Unix-like OS, those commands are witten in a Unix-like command interpreter (namely *zsh*) that are not directly compatible with Microsoft's *cmd* interpreter. This means that the initial steps of file ordering should be run by a Unix-like OS, opening a new future work of translating those commands to be used in Windows NT OS. After the ordering steps (that can be bypassed with variables `pre_ordered_GLM` and `pre_ordered_MMIA`), the only uso for the `os` library is to list directories entries, which based on `Python` documentation, just calls the native OS directory iteration system [9].

4.4 Budget

In this section a complete breakdown of the study's budget is given, detailing every major cost with a brief description.

As this study does not require any type of material or support apart from the computer, the major cost is given by the salary of a junior engineer, estimated as 15€/h. Tab.(4.1) shows a simple breakdown of the main work packages developed and the time dedicated to each one.

TABLE 4.1: Main work packages and cost associated

Work Package	***Dedication [h]***	***Unitary Cost [€/h]***	***Cost [€]***
Documentation	20	15	600
Code design	50	15	750
Code development	200	15	3000
Debugging and testing	100	15	1500
Code documentation	50	15	750
Runtime	20	0.015	0.45
		TOTAL:	6600.5€

Where documentation accounts for initial approach with the area, article consultation and coding package documentation. Design of the code accounts for the structure of the program, how steps are placed in order to achieve the final desired output. Code development and debbuging and testing packages are the longest by far as this study required a great dedication to developing the program. Runtime package is the cost derived from having a computer constantly plugged-in to compute operations while alone (some processes took long running times before some optimization). Finally, code documentation accounts for the development of this report as an introductory manual to the code as well as code presentation.

References

[1] E. A. Albrecht, R., *The 13 years of TRMM lightning imaging sensor: from individual flash characteristics to decadal tendencies*, (2011).

[2] O. Chanrion, T. Neubert, and et al., *The Modular Multispectral Imaging Array (MMIA) of the ASIM Payload on the International Space Station*, Space Science Reviews, 215 (2019).

[3] H. Christian and E. K. Aamodt, *Device for detecting an image of a non-planar surface*, (2011).

[4] S. Goodman, D. Buechler, K. Knupp, K. Driscoll, and E. McCaul, *The 1997–98 El Nino event and related wintertime lightning variations in the southeastern United States*, 4 (2000), pp. 541–554.

[5] S. J. Goodman, R. J. Blakeslee, W. J. Koshak, D. Mach, J. Bailey, D. Buechler, L. Carey, C. Schultz, M. Bateman, E. McCaul, and G. Stano, *The GOES-R Geostationary Lightning Mapper (GLM)*, Atmospheric Research, 125-126 (2013).

[6] T. Neubert, N. Østgaard, V. Reglero, E. Blanc, O. Chanrion, C. A. Oxborrow, A. Orr, M. Tacconi, O. Hartnack, and D. D. Bhanderi, *The ASIM Mission on the International Space Station*, Space Science Reviews, 215 (2019).

[7] NOAA, *Instruments: Geostationary Lighting Mapper (GLM).*

[8] NumPy, *numpy.convolve documentation page.*

[9] Python, *Python os library Documentation.*

[10] SciPy, *scipy.signal.correlate Documentation.*

[11] UniData, *Network Common Data Form (NetCDF).*

Appendices

A Code of the `main_TFG.py` script

Listing A.1: Full code for the `main_TFG.py` script of the program

```
"""
README

This script extracts data from GLM .nc files and MMIA .cdf files and prepares
it for comparison to extract detection sensitivity of GLM.

To do so, a time snippet for every day with GLM and MMIA files is extracted
to use only instead of the whole signal vectors using LINET's data from a
.csv file, where date, time, latitude, longitude, trigger ID and group_ID
are taken.

Every file from a given path 'GLM_files_path' (for GLM files) and
'MMIA_files_path' (for MMIA files) is classified in a different directory
according to the snippet (given by LINET's data) if MMIA, or date (day) of
the event if GLM, and stored in given paths 'GLM_ordered_dir' and
'MMIA_Ordered_dir' (for GLM and MMIA, respectively).

                            CAUTION!
This step needs to be run by a UNIX-like OS, as explicit terminal commands
are given using Python's library "os".

As the files have been ordered, data extraction is done using a Python
script for GLM and a MatLab script for MMIA, both given by Jesús López, only
for the matching dates between the existing GLM and MMIA files. This main
script initialises the MatLab engine by its own. This extracted data is
stored in different snippet .txt files for GLM and .mat files for MMIA at
given paths 'GLM_ordered_outputs' and 'MMIA_ordered_outputs'.

                            IMPORTANT
Extracting data from GLM's .nc files and MMIA's .cdf files takes a lot of time
and is a one-time step, specially for MMIA .cdf files where a MatLab engine
has to be started for every snippet. Because of that, one can set special
parameters 'pre_extracted_GLM' and 'pre_extracted_MMIA' to '0' to make that
```

```
extraction once, and then set those parameters to '1' to bypass the extracting
operations as the data has already been stored in '..._ordered_outputs'.
Again, this process needs to be run by a UNIX-like OS.

This data is then uploaded to the Python workspace and treated to allow
a comparison analysis using cross-correlation.

For GLM data, an integration of radiance is made every 0.002s (GLM sample rate)
and time and signal vectors are accommodated to MMIA sample rate of 0.00001s.

For MMIA data, afilter is applied to the 777.4nm photometer data vector in
order to reduce signal noise, and again accommodated to MMIA sample rate
in case any time jump was there.

Both types of data are then normalised and cross-correlated to get the time
shift between them, aligned and compared by counting peaks in their signals.

All functions are imported from 'TFG_library.py' to make this script simple
and clean.


@ Jaime Francisco Morán Domínguez, 2021

"""

import TFG_library as TFG
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import pickle

# Just for plot presentation in LaTeX Style (slows the program)
plt.rc('font', **{'family': 'serif', 'serif': ['latin modern roman']})



'''
################################################
##                 USER INPUT DATA            ##
################################################
'''

### GENERAL ###

# Boolean variable for pre-ordered files
```

```
pre_ordered_GLM = 1
pre_ordered_MMIA = 1

# Boolean variable for pre-extracted files
pre_extracted_GLM = 1
pre_extracted_MMIA = 1

# Boolean variable for generating plots
show_plots = 0


### LINET ###

# CAUTION!! Make sure to write the 'r' before the path string
    # Example: LINET_path = r'path_to_csv_file'
LINET_path = r'/Users/jaimemorandominguez/Desktop/Final/MMIA_match_pos_LINET_2020.csv'

# Time in seconds to analyze GLM and MMIA before and after LINET's time snippet
    # Recommended 0.2
cropping_margin = 0.15

# Plus of angle in latitude and longitude to snip GLM data (with respect to LINET data)
    # Recommended 0.5
angle_margin = 0.5


### GLM ###

# Path where GLM's .nc files are located
GLM_files_path = '/Users/jaimemorandominguez/Desktop/Final/GLM_archivos/nc'

# Path where you want your daily ordered GLM's .nc files to be located
GLM_ordered_dir = '/Users/jaimemorandominguez/Desktop/Final/GLM_archivos/Dairy_dir'

# Path where you want your daily ordered extracted GLM's .txt files to be located
GLM_ordered_outputs = '/Users/jaimemorandominguez/Desktop/Final/GLM_archivos/GLM_output'


### MMIA ###

# Path where MMIA's .cdf files are located
MMIA_files_path = '/Users/jaimemorandominguez/Desktop/Final/MMIA_archivos/cdf'

# Path where you want your snippet ordered MMIA's .cdf files to be located
MMIA_ordered_dir = '/Users/jaimemorandominguez/Desktop/Final/MMIA_archivos/MMIA_dairy'
```

```

# Path where you want your snippet ordered extracted MMIA's .mat
    files to be located
MMIA_ordered_outputs = '/Users/jaimemorandominguez/Desktop/Final/
    MMIA_archivos/MMIA_output'


'''
##################################################
##            END OF USER INPUT DATA            ##
##################################################
'''



########### LINET'S DATA UPLOAD ###########

# Uploading Linet's data from .csv

print(' ')
print("Uploading Linet's data...")
linet_data = pd.read_csv (LINET_path)
linet_data = linet_data.to_numpy()
print('Done!')
print(' ')


########### DAILY ORDERING GLM FILES ###########

if pre_ordered_GLM == 0:

    # Ordering GLM .nc files in dairy directories
    TFG.GLM_data_ordering(GLM_files_path, GLM_ordered_dir)

else:
    print('All GLM .nc files are already daily ordered')
    print(' ')


########### CHECKING FOR MATCHING DATES ###########

MMIA_dates= TFG.get_MMIA_dates(MMIA_files_path)

# Searching for dates that both GLM and MMIA have data from
matches = TFG.check_existance(GLM_ordered_dir, MMIA_dates)


########### EXTRACTING SNIPPET DATA FROM LINET'S DATAFRAME
    ###########

# Getting times from LINET data
[linet_times,indx] = TFG.get_LINET_timing(linet_data, matches)
del linet_data

###### MMIA'S DATA ORDERING, EXTRACTION, UPLOAD AND CONDITIONING
    ######
```

```

# Ordering MMIA data files into daily snippet folders
if pre_ordered_MMIA == 0:

    TFG.MMIA_data_ordering(MMIA_files_path, MMIA_ordered_dir, linet_times, matches)
else:
    print('All MMIA .cdf files are already daily ordered')
    print(' ')

# Extracting MMIA data into dairy .mat files
    # CAUTION! Only matching dates with GLM are being extracted
if pre_extracted_MMIA == 0:

    TFG.extract_MMIA(MMIA_files_path, MMIA_ordered_dir, MMIA_ordered_outputs, matches, linet_times)

else:
    print('All data from MMIA .cdf files has already been extracted')
    print(' ')

# Unifying all data in a structure of lists
MMIA_raw_data = TFG.unify_MMIA_data(MMIA_ordered_outputs, linet_times, matches, show_plots)

# Conditioning MMIA data for further analysis
MMIA_filtered = TFG.condition_MMIA_data(MMIA_raw_data, matches, show_plots)

del MMIA_raw_data


########### GLM'S DATA EXTRACTION, UPLOAD AND CONDITIONING ###########

# Extracting GLM data into dairy .txt files
    # CAUTION! Only snippets with MMIA data are being extracted
if pre_extracted_GLM == 0:

    TFG.extract_GLM(GLM_ordered_dir, GLM_ordered_outputs, linet_times, matches, MMIA_filtered, angle_margin, cropping_margin)

else:
    print('All data from GLM .nc files has already been extracted')
    print(' ')

del linet_times

# Unifying all data in a structure of matrices
GLM_raw_data = TFG.unify_GLM_data(GLM_ordered_outputs, MMIA_filtered, matches, show_plots)


# Conditioning GLM data for further analysis
```

```python
GLM_data = TFG.condition_GLM_data(GLM_raw_data, matches, show_plots)
del GLM_raw_data


########### CROSS-CORRELATION ###########

# Normalizing GLM data to cross-correlate with MMIA data
print('Normalizing GLM data...')
GLM_norm = [None] * len(GLM_data)
for i in range(len(GLM_data)):
    snip = [None] * len(GLM_data[i])
    GLM_norm[i] = snip

for i in range(len(GLM_data)):
    for j in range(len(GLM_data[i])):
        if type(GLM_data[i][j]) == np.ndarray:
            snippet = np.zeros((len(GLM_data[i][j]),2))
            GLM_norm[i][j] = snippet
            GLM_norm[i][j][:,0] = GLM_data[i][j][:,0]
            GLM_norm[i][j][:,1] = TFG.normalize(GLM_data[i][j][:,1])
print('Done!')
print(' ')

# Normalizing MMIA data to cross-correlate with GLM data
print('Normalizing MMIA data...')
MMIA_norm = [None] * len(MMIA_filtered)
for i in range(len(MMIA_filtered)):
    snip = [None] * len(MMIA_filtered[i])
    MMIA_norm[i] = snip

for i in range(len(MMIA_filtered)):
    for j in range(len(MMIA_filtered[i])):
        if type(MMIA_filtered[i][j]) == np.ndarray:
            snippet = np.zeros((len(MMIA_filtered[i][j]),2))
            MMIA_norm[i][j] = snippet
            MMIA_norm[i][j][:,0] = MMIA_filtered[i][j][:,0]
            MMIA_norm[i][j][:,1] = TFG.normalize(MMIA_filtered[i][j][:,1])
print('Done!')
print(' ')

# Cross-correlating snippets
show_plots = 1
[GLM_xcorr, MMIA_xcorr, delays] = TFG.cross_correlate_GLM_MMIA(GLM_data, MMIA_filtered, GLM_norm, MMIA_norm, matches, show_plots)
del GLM_data
del MMIA_filtered

# Saving cross-correlated data
f = open('xcorr_data.pckl', 'wb')
pickle.dump([GLM_xcorr,MMIA_xcorr, GLM_norm, MMIA_norm, matches, delays], f)
f.close()
```

```
del GLM_norm
del MMIA_norm

# Getting peaks from cross-correlated signals
[GLM_peaks, MMIA_peaks] = TFG.get_GLM_MMIA_peaks(GLM_xcorr,
    MMIA_xcorr, matches, show_plots)

# Getting delay statistics
[total_snippets, avg_delay, avg_abs_delay, avg_MMIA_delay,
    std_MMIA_delay, avg_GLM_delay, std_GLM_delay, MMIA_delays,
    GLM_delays, no_delays] = TFG.study_delays(delays, GLM_xcorr,
    MMIA_xcorr, show_plots)
```

B Code for important functions in order of appearance in main_TFG.py

B.1 Function GLM_data_ordering.py

Listing B.1: Full code for GLM_data_ordering.py function

```
def GLM_data_ordering(read_path, dir_path):
    '''
    This function gets a directory where unordered .nc files are located
    and gets them daily ordered in diferent directories inside dir_path.
    Those new directories are named by year-month-day.

    Parameters
    ----------
    read_path : string
        Path of the directory where the unordered .nc files are located.
    dir_path : string
        Path to the directory where the new daily-ordered directories with
        ordered separated .nc files will be located.

    Returns
    -------
    None. Daily-ordered GLM files in new directories.
    '''

    print('Ordering GLM data into separate dates...')
    with os.scandir(read_path) as files:
        files = [file.name for file in files if file.is_file() and file.name.endswith('.nc')]
    if len(files)==0:
        print('Error: No GLM .nc files found to process!')

    for i in range(len(files)):
        date = datetime.datetime.strptime(files[i][44:48] + "-" + files[i][48:51], "%Y-%j").strftime("%Y%m%d")
        if i==0:         # First file does not have any folder yet
            os.system('mkdir ' + dir_path+'/'+date)
            os.system('cp ' + read_path+'/'+files[0] + ' ' + dir_path+'/'+date)
        else:            # All the other files
            subfolders = [f.path for f in os.scandir(dir_path) if f.is_dir()]
            if subfolders.count(date)==0:   # If there is not a folder with that date
                os.system('mkdir '+dir_path+'/'+date)
```

```
                os.system('cp ' + read_path+'/'+files[i] + ' ' + dir_path+'/'+date)
            else:       # If there already exists a folder with that date
                os.system('cp ' + read_path+'/'+files[i] + ' ' + dir_path+'/'+date)
    print('Done')
    print(' ')
```

B.2 Function get_MMIA_dates.py

Listing B.2: Full code for get_MMIA_dates.py function

```
def get_MMIA_dates(read_path):
    '''
    This function just hovers over MMIA .cdf files to extract a list with
    all existing dates with MMIA data.

    Parameters
    ----------
    read_path : string
        Path to the directory where all MMIA .cdf files are stored.

    Returns
    -------
    MMIA_dates : list
        List of strings with all dates with existing MMIA data, in the form
        YearMonthDay.
    '''

    print('Getting the list of MMIA dates with existing data..')

    with os.scandir(read_path) as files:
        files = [file.name for file in files if file.is_file() and file.name.endswith('.cdf')]
    if len(files)==0:
        print('Error: No MMIA .cdf files found to process!')

    MMIA_dates = []

    for i in range(len(files)):
        date = files[i][50:54]+files[i][55:57]+files[i][58:60]

        if i==0:              # First file does not have any existing date
            MMIA_dates.append(date)
        else:                 # All the other files
            if MMIA_dates.count(date) == 0: # If there is no register of that date
                MMIA_dates.append(date)

```

```
    print('Done')
    print(' ')

    return MMIA_dates
```

B.3 Function check_existance.py

LISTING B.3: Full code for check_existance.py function

```
def check_existance(GLM_ordered_dir, MMIA_dates):
    '''
    This function determines which dates are present among both GLM and MMIA
    sets of data. It returns a vector with the matching dates as strings in
    the form YearMonthDay.

    Parameters
    ----------
    GLM_ordered_dir : str
        Path to the directory where daily ordered GLM .nc files are located.
    MMIA_dates : array
        Vector containing all the dates in the MMIA's set of data.

    Returns
    -------
    matches : array
        Vector containing just the dates that are present in GLM and MMIA
        sets of data.
    '''

    print('Checking for matching dates between GLM and MMIA data...')
    print(' ')

    GLM_dates = os.listdir(GLM_ordered_dir)

    if GLM_dates.count('.DS_Store')!=0:
        GLM_dates.remove('.DS_Store')

    GLM_len = len(GLM_dates)
    MMIA_len = len(MMIA_dates)
    matches = []

    if GLM_len <= MMIA_len:
        for i in range(GLM_len):
            if GLM_dates[i] in MMIA_dates:
                matches.append(GLM_dates[i])

    elif MMIA_len < GLM_len:
        for i in range(MMIA_len):
```

```
            if MMIA_dates[i] in GLM_dates:
                matches.append(MMIA_dates[i])

    if len(matches) == GLM_len and len(matches) == MMIA_len:
        print('Both GLM and MMIA data correspond to same days. All data will be analyzed')
    elif len(matches) < GLM_len and len(matches) < MMIA_len:
        print('There is/are just %d matching date/s, which will be analyzed. Both GLM and MMIA daily ordered folders contain more dates that do not match each other' % len(matches))
    elif len(matches) == GLM_len and len(matches) < MMIA_len:
        print('All GLM dates correspond to MMIA dates and will be analyzed. The folder where MMIA daily ordered folders are located contais %d more date/s than GLMs' % (MMIA_len-len(matches)))
    elif len(matches) == MMIA_len and len(matches) < GLM_len:
        print('All MMIA dates correspond to GLM dates and will be analyzed. The folder where GLM daily ordered folders are located contais %d more date/s than MMIAs' % (GLM_len-len(matches)))
    print(' ')

    return matches
```

B.4 Function get_linet_timing.py

Listing B.4: Full code for get_linet_timing.py function

```
def get_LINET_timing(linet_data, matches):
    '''
    This function gets the data from linet_data and generates a list of lists
    where information of different events is stored only for dates with
    existing GLM and MMIA data. Date order is the same as in matches.

    Parameters
    ----------
    linet_data : matrix
        Linet's data from the .csv file ordered in a numpy array.
    matches : list
        List of dates with existing GLM and MMIA data files.

    Returns
    -------
    events : list
        List of lists containing different events for every existing GLM
        and MMIA data. For every day layer, the structure is:
        Events structure (for every day layer):
           1st column: Event starting time
           2nd column: Event ending time
           3rd column: Event min latitude
```

```
            4th column: Event max latitude
            5th column: Event min longitude
            6th column: Event max longitude
            7th column: List of MMIA trigger ID's in this snippet
    '''

    print('Getting time snippets from LINET data...')

    # Getting only dates with existing GLM and MMIA data
    linet_subset = []
    # linet_subset structure:
        #   1st column: Event date
        #   2nd column: Event hour
        #   3rd column: Event latitude
        #   4th column: Event longitude
        #   5th column: Event ID
        #   6th column: Group ID

    for i in range(len(linet_data)):
        date = datetime.datetime.strptime(linet_data[i,0][0:10], "%Y-%m-%d").strftime("%Y%m%d")
        if date in matches:
            hour = float(linet_data[i,0][11:13])*3600 + float(linet_data[i,0][14:16])*60 + linet_data[i,1]
            linet_subset.append([date, hour, linet_data[i,3], linet_data[i,4], linet_data[i,6], linet_data[i,7]])
    if len(linet_subset) == 0:
        print('Your .csv file does not contain any data of the dates of existing GLM and MMIA data files')
    else:
        # Extracting times from linet data, generation of days:
        events = []
        for i in range(len(matches)):
            events.append([])
        # events structure (per day layer):
            #   1st column: Event min hour
            #   2nd column: Event max hour
            #   3rd column: Event min latitude
            #   4th column: Event max latitude
            #   5th column: Event min longitude
            #   6th column: Event max longitude
            #   7th column: Event MMIA ID's

        # Extracting data from those lines with group ID != 0
        linet_subset_gID = []
        gIDs = []
        no_gID = []
        for i in range(len(linet_subset)):
            if linet_subset[i][5] != 0: # If group ID is not 0 (no data)
                linet_subset_gID.append(linet_subset[i])
                if i == 0:
                    gIDs.append(linet_subset[i][5])
                else:
                    if gIDs.count(linet_subset[i][5]) == 0:
                        gIDs.append(linet_subset[i][5])
```

```
        else:
            no_gID.append(i)

    # Removing all lines with group_ID from linet_subset
    linet_subset = [linet_subset[index] for index in no_gID]


    # Checking for wrongly repeated Group ID's
    indx = []
    for i in range(len(gIDs)):
        indx.append([])

    for i in range(len(indx)): # Creating lists of indexes with same GroupID
        for j in range(len(linet_subset_gID)):
            if linet_subset_gID[j][5] == gIDs[i]:
                indx[i].append(j)

    for i in range(len(indx)): # Checking incorrect index jumps
        if len(indx[i]) > 1:
            jumps = []
            for j in range(1,len(indx[i])):
                if indx[i][j] > indx[i][j-1]+5:
                    jumps.append(j)
            if len(jumps) != 0:
                new_gIDs = [None] * len(jumps)
                new_groups = [None] * len(jumps)
                for k in range(len(jumps)):
                    new_gIDs[k] = int(str(i)+str(k))*100000 # Assign a non-existent GroupID of 6 digits
                    gIDs.append(new_gIDs[k])
                    if k != len(jumps)-1:
                        new_groups[k] = indx[i][jumps[k]:jumps[k+1]]
                    else:
                        if jumps[k] < len(indx[i])-1:
                            new_groups[k] = indx[i][jumps[k]:-1]
                            new_groups[k].append(indx[i][-1])
                        else:
                            new_groups[k] = [indx[i][-1]]
                for k in range(len(new_groups)):
                    for m in range(len(new_groups[k])):
                        linet_subset_gID[new_groups[k][m]][5] = new_gIDs[k]

    day_per_gID = []
    hour_per_gID = []
    lat_per_gID = []
    lon_per_gID = []
    ID_per_gID = []
    for i in range(len(gIDs)):
        day_per_gID.append([])
        hour_per_gID.append([])
        lat_per_gID.append([])
        lon_per_gID.append([])
```

```python
        ID_per_gID.append([])

    for i in range(len(linet_subset_gID)):
        pos = gIDs.index(linet_subset_gID[i][5])
        day_per_gID[pos] = linet_subset_gID[i][0]
        hour_per_gID[pos].append(linet_subset_gID[i][1])
        lat_per_gID[pos].append(linet_subset_gID[i][2])
        lon_per_gID[pos].append(linet_subset_gID[i][3])
        ID_per_gID[pos].append(linet_subset_gID[i][4])

    for i in range(len(gIDs)):
        day_pos_in_matches = matches.index(day_per_gID[i])
        list_to_events = [min(hour_per_gID[i]), max(hour_per_gID[i]), min(lat_per_gID[i]), max(lat_per_gID[i]), min(lon_per_gID[i]), max(lon_per_gID[i]), ID_per_gID[i]]
        events[day_pos_in_matches].append(list_to_events)


    # Extracting data from those lines with group ID == 0
    event_pos = [None] * len(matches)
    # Accounting for positions taken by lines with group_ID != 0
    for i in range(len(events)):
        event_pos[i] = len(events[i])

    row_0 = [linet_subset[0][1], linet_subset[0][1], linet_subset[0][2], linet_subset[0][2], linet_subset[0][3], linet_subset[0][3], [linet_subset[0][4]]]

    events[matches.index(linet_subset[0][0])].append(row_0)

    for i in range(1,len(linet_subset)):

        current_day = linet_subset[i][0]
        prev_day = linet_subset[i-1][0]
        same_day = current_day == prev_day
        day_pos = matches.index(current_day)

        current_hour = linet_subset[i][1]
        prev_hour = linet_subset[i-1][1]
        same_hour = prev_hour <= current_hour and current_hour <= prev_hour + 1

        current_lat = linet_subset[i][2]

        current_lon = linet_subset[i][3]

        current_ID = linet_subset[i][4]

        same_event = [same_day, same_hour]

        if all(same_event) == True:
            # Refresh end time
            events[day_pos][event_pos[day_pos]][1] = current_hour
            # Refresh max and min latitude if necessary
```

```python
                if current_lat < events[day_pos][event_pos[day_pos]][2]:
                    events[day_pos][event_pos[day_pos]][2] = current_lat
                elif current_lat > events[day_pos][event_pos[day_pos]][2]:
                    events[day_pos][event_pos[day_pos]][3] = current_lat
                # Refresh max and min longitude if necessary
                if current_lon < events[day_pos][event_pos[day_pos]][4]:
                    events[day_pos][event_pos[day_pos]][4] = current_lon
                elif current_lon > events[day_pos][event_pos[day_pos]][5]:
                    events[day_pos][event_pos[day_pos]][5] = current_lon
                # Add trigger ID to snippet list of ID's
                events[day_pos][event_pos[day_pos]][6].append(current_ID)

            else:
                events[day_pos].append([current_hour, current_hour, current_lat, current_lat, current_lon, current_lon, [current_ID]])
                if same_day == True:
                    event_pos[day_pos] = event_pos[day_pos] + 1

    print('Done!')
    print(' ')
    return [events,indx]
```

B.5 Function MMIA_data_ordering.py

Listing B.5: Full code for MMIA_data_ordering.py function

```python
def MMIA_data_ordering(read_path, dir_path, linet_times, matches):
    '''
    This function gets a directory where unordered .cdf files are located
    and gets them snippet-ordered in diferent directories inside dir_path.
    Those new directories are named by YearMonthDay_snippetIndex, and files
    are named after their ID number.

    Parameters
    ----------
    read_path : string
        Path of the directory where the unordered .cdf files are located.
    dir_path : string
```

```
        Path to the directory where the new daily-ordered directories with
        ordered separated .cdf files will be located.
    linet_times : list_to_events
        List of daily lists of snippet info (time, geolocation and MMIA Id's)
    matches : list
        List of dates with existing GLM and MMIA files.

    Returns
    -------
    Snippet-ordered files in new directories.
    '''

    print('Ordering .cdf files according to ID in snippet...')

    with os.scandir(read_path) as files:
        files = [file.name for file in files if file.is_file() and file.name.endswith('.cdf')]

    to_analize = []
    for i in range(len(files)):
        file_date = files[i][50:54] + files[i][55:57] + files[i][58:60]
        if matches.count(file_date) != 0: # Exists in matches
            to_analize.append(i)
    files = [files[index] for index in to_analize]

    if len(files)==0:
        print('Error: No matching MMIA .cdf files found to process!')

    ID_path = read_path+'/cdf_ID'
    os.system('mkdir ' + ID_path)

    for i in range(len(files)):
        if len(files[i]) == 92:
            ID = files[i][85:88]
        elif len(files[i]) == 93:
            ID = files[i][85:89]
        elif len(files[i]) == 94:
            ID = files[i][85:90]
        elif len(files[i]) == 108:
            ID = files[i][85:88]
        elif len(files[i]) == 109:
            ID = files[i][85:89]
        elif len(files[i]) == 110:
            ID = files[i][85:90]

        os.system('cp ' + read_path+'/'+files[i] + ' ' + ID_path+'/'+ID+'.cdf')

    for i in range(len(linet_times)):
        for j in range(len(linet_times[i])):
            os.system('mkdir '+dir_path+'/'+matches[i]+'_'+str(j))
            IDs = linet_times[i][j][-1]
```

```
            for k in range(len(IDs)):
                os.system('mv '+ID_path+'/'+str(IDs[k])+'.cdf' + ' ' + dir_path+'/'+matches[i]+'_'+str(j))
    os.system('rm -rf '+ID_path)
    print('Ordering done!')
    print(' ')
```

B.6 Function extract_MMIA.py

Listing B.6: Full code for extract_MMIA.py function

```
def extract_MMIA(MMIA_files_path, MMIA_ordered_dir, MMIA_ordered_outputs, matches, linet_times):
    '''
    This function gets the ordered .cdf files and calls a MatLab script
    to process them, returning the .mat files to an output folder whose
    names match the file's date.

    Parameters
    ----------
    MMIA_files_path : string
        Path to the directory where the ordered .cdf files are located.
    MMIA_ordered_outputs : string
        Path to the directory where the snippet-ordered files are located.
    MMIA_ordered_outputs : string
        Path to the directory where the processed files will be located.
    matches : list
        List of common dates with GLM and MMIA data, as strings in the form
        YearMonthDay.
    linet_times : list
        List of daily lists of important snippet information.

    Returns
    -------
    Snippet-ordered and extracted MMIA .cdf files into a folder with
    .mat extension.
    '''

    print('Extracting data from MMIA .cdf files...')
    print(' ')

    for i in range(len(linet_times)):

        print('Processing MMIA data, date %d %d / %d...' % (int(matches[i]), i+1, len(matches)))

```

```
        for j in range(len(linet_times[i])):

            current_snippet_path = MMIA_ordered_dir + '/' + matches[i] + '_' + str(j)

            with os.scandir(MMIA_ordered_outputs) as pre_done:
                pre_done = [file.name for file in pre_done if file.is_file() and file.name.endswith('.mat')]

            if pre_done.count(matches[i] + '_' + str(j)+'.mat') == 0:

                with os.scandir(current_snippet_path) as files:
                    files = [file.name for file in files if file.is_file() and file.name.endswith('.cdf')]
                size = len(files)

                if size != 0:
                    print('Starting the MatLab engine and extracting data from .cdf files for snippet %d, %d / %d...' % (int(matches[i]),j, len(linet_times[i])))
                    eng = matlab.engine.start_matlab()
                    path = MMIA_ordered_dir + '/' + matches[i] + '_' + str(j) + '/'
                    eng.workspace['str'] = path
                    eng.MMIA_symplified_v4(nargout=0)
                    eng.quit()
                    wd = os.getcwd()
                    os.system('mv '+wd+'/MMIA_data.mat ' + MMIA_ordered_outputs+'/'+matches[i]+'_' + str(j) +'.mat')
                else:
                    print('There is no MMIA data for date %d, snippet %d' % (int(matches[i]), j))
            else:
                print('MMIA data for day %d snippet %d was pre-extracted' % (int(matches[i]), j))
        print('Date',matches[i],' done')
        print(' ')
    print('Your  processed .mat files can be accessed at ', MMIA_ordered_outputs)
    print(' ')
```

B.7 Function unify_MMIA_data.py

Listing B.7: Full code for unify_MMIA_data.py function

```
def unify_MMIA_data(output_path, linet_times, matches, show_plots):
    '''
    This functions gets all the MMIA's extracted data .mat files
    from the directory output_path and creates and returns list MMIA_raw_data.

    Parameters
```

```
    ----------
    output_path : string
        Path to the existing directory where the resulting daily .mat files
        are located.
    linet_times : list
        List of daily lists of important snippet information.
    matches : list
        List of common dates with GLM and MMIA data, as strings in the form
        YearMonthDay.
    show_plots : bool
        Boolean variable for showing plots all through the program.

    Returns
    -------
    MMIA_data : list
        A list of tables with all the information fount in the .mat files.
    '''

    # Creation of the list of lists for MMIA data
    MMIA_raw_data = [None] * len(linet_times)
    for i in range(len(linet_times)):
        snippets = [None] * len(linet_times[i])
        MMIA_raw_data[i] = snippets

    with os.scandir(output_path) as files:
        files = [file.name for file in files if file.is_file() and file.name.endswith('.mat')]
    size = len(files)
    if size == 0:
        print('No MMIA .mat files found!')

    print('Uploading MMIA data from .mat files...')
    print(' ')

    for i in range(size):
        current_path = output_path + '/' + files[i]
        day = files[i][0:8]
        day_pos = matches.index(day)
        ID = int(files[i][9:-4])

        mat = sio.loadmat(current_path)
        current_data = mat.get('MMIA_all')

        time_jump = 0
        min_jump_time = 5 # [s]
        pos = 1
        while time_jump == 0:
            # If there is a time jump
            if current_data[pos,0]-current_data[pos-1,0] >= min_jump_time:
                time_jump = 1
            # If there is no time jump and position is not last
```

```
            elif current_data[pos,0]-current_data[pos-1,0] < min_jump_time and pos != len(current_data)-1:
                pos = pos+1
            # If there is no time jump and position is last
            elif current_data[pos,0]-current_data[pos-1,0] < min_jump_time and pos == len(current_data)-1:
                time_jump = 2

        if time_jump == 1:
            print('Day %s snippet %d has a time jump!' % (day, ID))
            print("This means that two or more lines in Linet's .csv have the same ID while they do not belong to the same group")
            current_data = None

        MMIA_raw_data[day_pos][ID] = current_data

        if show_plots == 1 and time_jump !=1:
            plt.figure()
            plt.plot(MMIA_raw_data[day_pos][ID][:,0],MMIA_raw_data[day_pos][ID][:,1],linewidth=0.1, color='r')
            plt.title('MMIA 777.4nm Photometer Detections for day %d snippet %d with no filter applied' % (int(day), ID))
            plt.xlabel('Time [s]')
            plt.ylabel('Energy')
            plt.grid('on')
            plt.show()

    print('Done!')
    print(' ')
    return MMIA_raw_data
```

B.8 Function condition_MMIA_data.py

LISTING B.8: Full code for condition_MMIA_data.py function

```
def condition_MMIA_data(MMIA_data, matches, show_plots):
    '''
    This functions takes 'MMIA_data', a list of MMIA tables of information
    and applies a filter in 777.4nm photometer information vector to reduce
    noise.
    It also plots every signal with and without the filter applied
    if 'show_plots' is True.

    Parameters
    ----------
    MMIA_data : list
        List of MMIA tables of information.
    matches : list
        List of common dates with GLM and MMIA data, as strings in the form
```

```
        YearMonthDay.
    show_plots : bool
        Boolean variable for plotting. Not ploting makes the program faster.

    Returns
    -------
    MMIA_filtered : list
        A list of MMIA tables of information with a filter applied
        and only regarding time and 777.4nm photometer information
    '''

    # Creation of the new list of lists of MMIA data with Moving Average
    MMIA_filtered = [None] * len(MMIA_data)
    for i in range(len(MMIA_data)):
        snippets = [None] * len(MMIA_data[i])
        MMIA_filtered[i] = snippets

    for i in range(len(MMIA_data)): # For every day of MMIA data
        print('Date %d, %d / %d...' % (int(matches[i]), i+1, len(matches)))
        for j in range(len(MMIA_data[i])):

            if type(MMIA_data[i][j]) == np.ndarray:

                print('Applying a filter to reduce noise to MMIA signal, date %d snippet %d / %d...' % (int(matches[i]), j, len(MMIA_data[i])))

                current_data=np.zeros((len(MMIA_data[i][j]),2))
                current_data[:,0] = MMIA_data[i][j][:,0]
                current_data[:,1] = MMIA_data[i][j][:,1]


                if show_plots == 1:
                    plt.figure(figsize=(9, 3))
                    plt.plot(current_data[:,0],current_data[:,1], linewidth=0.5, color='r')
                # ------------------------------------------------------------------------------

                n = 15  # the larger n is, the smoother curve will be
                b = [1.0 / n] * n
                a = 1
                current_data[:,1] = lfilter(b,a,current_data[:,1])

                if (current_data[:,1] < 1.75).all() == True:
                    MMIA_filtered[i][j] = None
                    print('MMIA data for day %d snippet %d was just noise!' % (int(matches[i]), j))
                    print(' ')
                else:

                    # Assuring continuity in MMIA_MA timesteps
```

```
                        MMIA_filtered[i][j] = TFG.fit_vector_in_MMIA_timesteps(current_data, int(matches[i]), j, 1, 1)
                        #MMIA_filtered[i][j] = current_data

                        if show_plots == 1:
                            # MMIA representation with filter and with rectified time
                            #plt.figure()
                            plt.plot(MMIA_filtered[i][j][:,0], MMIA_filtered[i][j][:,1],linewidth=0.5, color='b')
                            plt.title("Untreated (red) and filtered (blue) MMIA 777.4nm photometer detections of day %d snippet %d" % (int(matches[i]), j))
                            plt.xlabel('Time [s]')
                            plt.grid('on')
                            plt.ylabel(r"Irradiance $\left[\dfrac{\mu W}{m^2}\right]$")
                            plt.legend(['Untreated signal', 'Filtered signal'])
                            plt.show()

                else:
                    print('There is no MMIA data for day %d, snippet %d' % (int(matches[i]), j))
                    print(' ')

    print('Done!')
    print(' ')
    return MMIA_filtered
```

B.9 Function extract_GLM.py

Listing B.9: Full code for extract_GLM.py function

```
def extract_GLM(dir_path, output_path, linet_times, matches, MMIA_MA, angle_margin, cropping_margin):
    '''
    This function calls every directory with .nc files and extracts the data
    of all the files in it via GLM_processing function.

    Parameters
    ----------
    dir_path : string
        Path to the directory where the daily-ordered directories with
        ordered separated .nc files are located.
    output_path : string
        Path to the existing directory where the resulting daily .txt files
        will be located.
    linet_times : list_to_events
```

```
        List of daily lists of snippet info (time, geolocation and MMIA Id's)
    matches : list
        List of dates with existing GLM and MMIA files
    MMIA_MA : list
        List of daily lists of MMIA's time and signal (with a filter
        applied) vectors for every snippet
    angle_margin : float
        Plus of latitude and longitude angle for extracting GLM data with
        respect to Linet's data.
    cropping_margin : float
        Plus of time before and afer MMIA snippet times (or Linet times) for
         extracting GLM data.

    Returns
    -------
    .txt files for every snippet with GLM data prepared to be analyzed.
    '''

    print('Extracting data from GLM .nc files into snippet .txt...')
    print(' ')

    for i in range(len(linet_times)):  # Analyzing each directory's .nc files
        for j in range(len(linet_times[i])):

            if type(MMIA_MA[i][j]) == np.ndarray:

                print('Extracting GLM data for date %d snippet %d...' % (int(matches[i]), j))

                min_lat = linet_times[i][j][2] - angle_margin
                max_lat = linet_times[i][j][3] + angle_margin

                min_lon = linet_times[i][j][4] - angle_margin
                max_lon = linet_times[i][j][5] + angle_margin

                start_time = MMIA_MA[i][j][0,0] - cropping_margin
                end_time = MMIA_MA[i][j][-1,0] + cropping_margin

                # If willing to use Linet's timing, uncomment below
                #start_time = linet_times[i][j][0] - cropping_margin
                #end_time = linet_times[i][j][1] + cropping_margin

                TFG.GLM_processing(dir_path+'/'+matches[i]+'/', output_path, matches[i]+'_'+str(j), min_lat, max_lat, min_lon, max_lon, start_time, end_time)

                print('Date %s snippet %d done' % (matches[i], j))
                print(' ')
```

```
            else:
                print('GLM data for date %d snippet %d will not be extracted due to lack of MMIA data' % (int(matches[i]), j))
                print(' ')

    print('Your processed .txt files can be accessed at ', output_path)
    print(' ')
```

B.10 Function unify_GLM_data.py

Listing B.10: Full code for unify_GLM_data.py function

```
def unify_GLM_data(output_path, MMIA_MA, matches, show_plots):
    '''
    This function gets all the GLM's extracted data .txt files from the
    directory output_path and creates and returns list GLM_data.

    Parameters
    ----------
    output_path : string
        Path to the existing directory where the resulting daily .txt files
        are located.
    MMIA_MA : list
        List of daily lists of MMIA's time and signal (with a moving average
        applied) vectors for every snippet
    matches : list
        List of dates with existing GLM and MMIA files
    show_plots : bool
        Boolean value for outputting plots all through the program.

    Returns
    -------
    GLM_data : list
        A list of daily lists with all the information found in the .txt files,
        ordered by snippets.
    '''

    print('Uploading GLM data from .txt files...')
    # Creation of a new list of daily GLM snippets
    GLM_data = [None] * len(MMIA_MA)
    for i in range(len(MMIA_MA)):
        snippets = [None] * len(MMIA_MA[i])
        GLM_data[i] = snippets

    with os.scandir(output_path) as files:
        files = [file.name for file in files if file.is_file() and file.name.endswith('.txt')]
    size = len(files)
```

```
    if size == 0:
        print('No GLM .txt files found!')

    column_subset = ['Time', 'Event_lat', 'Event_lon', 'Event_ID', 'Flash_lat', 'Flash_lon', 'Radiance']

    for i in range(size):

        day = files[i][0:8]
        day_pos = matches.index(day)
        snip = int(files[i][9:-4])

        current_path = output_path + '/' + files[i]

        # Uploading the GLM's .txt using Pandas to sort it by time
        current_data = pd.read_csv(current_path, names=column_subset, sep='\s+')
        # Sorting data by time
        current_data = current_data.sort_values(by='Time')
        # Translating Pandas Dataframe to Numpy Matrix for easy data access
        current_data = current_data.to_numpy()
        # Appending current day to GLM_data
        GLM_data[day_pos][snip] = current_data

        if show_plots == 1:
            # Showing event map
            plt.figure()
            plt.scatter(current_data[:,2],current_data[:,1])
            plt.scatter(current_data[:,5],current_data[:,4],marker='x')
            plt.axis('equal')
            plt.grid('on')
            plt.title('Event grid for day %d snippet %d' % (int(day), snip))
            plt.xlabel('Longitude [deg]')
            plt.ylabel('Latitude [deg]')
            plt.show()
    print('Done!')
    print(' ')

    return GLM_data
```

B.11 Function condition_GLM_data.py

Listing B.11: Full code for condition_GLM_data.py function

```
def condition_GLM_data(GLM_total_raw_data, matches, show_plots):
    '''
    This function takes all the extracted data from GLM .txt files, integrates
    it and fits it in MMIA sample rate (0.002s of GLM to 0.00001s of MMIA)
```

```

    Parameters
    ----------
    GLM_total_raw_data : list
        List of daily GLM tables of data.
    matches : list
        List of dates with existing GLM and MMIA files
    show_plots : bool
        Boolean variable for plotting. Not ploting makes the program faster.

    Returns
    -------
    GLM_data : list
        List of daily lists of snippets with integrated GLM radiance in MMIA
        sample rate.
    '''

    # Creating a new set of data
    GLM_data = [None] * len(GLM_total_raw_data)
    for i in range(len(GLM_total_raw_data)):
        snippets = [None] * len(GLM_total_raw_data[i])
        GLM_data[i] = snippets

    # Integrating and extending GLM vectors by date
    for i in range(len(GLM_total_raw_data)):   # For every date with GLM data
        print('Integrating and extending GLM data vector for day %d...' % int(matches[i]))
        for j in range(len(GLM_total_raw_data[i])):

            if type(GLM_total_raw_data[i][j]) == np.ndarray and len(GLM_total_raw_data[i][j]) <= 1:
                print('GLM detection for day %d snippet %d is void! ' % (int(matches[i]), j))
                print(' ')
                GLM_total_raw_data[i][j] = None

            elif type(GLM_total_raw_data[i][j]) == np.ndarray and len(GLM_total_raw_data[i][j]) != 0:
                just_one_timestep = 1
                check_pos = 1
                while just_one_timestep == 1:
                    # If last position and same as timestep before in the .txt
                    if check_pos == (len(GLM_total_raw_data[i][j])-1) and GLM_total_raw_data[i][j][check_pos-1,0] == GLM_total_raw_data[i][j][check_pos,0]:
                        just_one_timestep = 2
                    else:
                        # Different timestep in the .txt as in line before
                        if GLM_total_raw_data[i][j][check_pos-1,0] != GLM_total_raw_data[i][j][check_pos,0]:
                            just_one_timestep = 0
```

```
                        else: # Same timestep in the .txt as in line before
                            check_pos = check_pos + 1

            if type(GLM_total_raw_data[i][j]) == np.ndarray and just_one_timestep == 2:
                print('GLM detection for day %d snippet %d contains only 1 timestep and will not be compared' % (int(matches[i]), j))
                print(' ')
                GLM_total_raw_data[i][j] = None

            if type(GLM_total_raw_data[i][j]) == np.ndarray:

                # Showing non-inflated time vector
                if show_plots == 1:
                    plt.figure()
                    plt.plot(GLM_total_raw_data[i][j][:,0])
                    plt.xlabel('Samples')
                    plt.ylabel('Time [s]')
                    plt.title('Original GLM Time VS Samples for date %d snippet %d' % (int(matches[i]), j))
                    plt.grid('on')
                    plt.show()

                # Integration

                print('Integrating GLM data, date %d snippet %d / %d' % (int(matches[i]), j, len(GLM_total_raw_data[i])))

                GLM_length = math.ceil((GLM_total_raw_data[i][j][-1,0] - GLM_total_raw_data[i][j][0,0]) / 0.002)

                # Current table of data (current day)

                GLM_int_data = np.zeros((GLM_length, 2))

                pos_0 = 0
                for k in range(GLM_length): # For every sample accounting zeroes at GLM rate
                    GLM_int_data[k,0] = round(GLM_total_raw_data[i][j][0,0] + k*0.002, 3)
                    t_min = GLM_int_data[k,0]
                    t_max = t_min + 0.002
                    inside = True
                    count = 0

                    while inside == True:
                        raw_pos = pos_0 + count

                        if GLM_total_raw_data[i][j][raw_pos,0] >= t_min and GLM_total_raw_data[i][j][raw_pos,0] < t_max:
                            GLM_int_data[k,1] = GLM_int_data[k,1] + GLM_total_raw_data[i][j][raw_pos,6]

```

```
                        # Check if the next GLM_total_raw_data sample will be added

                        raw_end = (raw_pos == (len(GLM_total_raw_data[i][j])-1))

                        if raw_end == False:
                            inside = (GLM_total_raw_data[i][j][raw_pos+1,0]) < t_max
                            if inside == True:
                                count = count + 1
                            else:
                                count = count + 1
                                pos_0 = raw_pos
                        else:
                            inside = False
                print('Done!')

                GLM_data[i][j] = fit_vector_in_MMIA_timesteps(GLM_int_data, int(matches[i]), j, show_plots, 0)

                # Check for too short snippet vectors
                if len(GLM_data[i][j])<=100:
                    print('Data for day %s snippet %d is too poor, only %d samples. This snippet will be omitted.' % (matches[i], j, len(GLM_data[i][j])))
                    GLM_data[i][j] = None

                if show_plots == 1 and type(GLM_data[i][j]) == np.ndarray:
                    # Plotting lineality in GLM time vector with GLM sampling rate
                    plt.figure()
                    plt.plot(GLM_int_data[:,0])
                    plt.title('GLM Time vector of day %d snippet %d with 0.002s period' % (int(matches[i]), j))
                    plt.xlabel('Samples')
                    plt.ylabel('Time [s]')
                    plt.grid('on')
                    plt.show()

                    # Integrated GLM radiance vs time graph representation
                    plt.figure()
                    plt.plot(GLM_int_data[:,0],GLM_int_data[:,1], linewidth=0.5, color='black')
                    plt.grid('on')
                    plt.title('GLM signal of day %d snippet %d with GLM sample rate (0.002s)' % (int(matches[i]), j))
                    plt.xlabel('Time [s]')
                    plt.ylabel('Radiance [J]')
                    plt.show()

    print('Integration of GLM vectors done!')
    print(' ')

```

```
    return GLM_data
```

B.12 Function fit_vector_in_MMIA_timesteps.py

Listing B.12: Full code for fit_vector_in_MMIA_timesteps.py function

```
def fit_vector_in_MMIA_timesteps(GLM_int_data, day, snippet, show_plots, is_MMIA):
    '''
    This function takes a time and signal pair of vectors and accommodates it
    into MMIA timesteps of 0.00001s. Inexistent values in between are filled
    by simple linear regression.

    Parameters
    ----------
    GLM_int_data : list
        List of daily lists of data snippets. NOT necessarily GLM data.
    day : int
        Date of the form YearMonthDay.
    snippet : int
        Index of the current snippet to expand inside day "day".
    show_plots : bool
        Boolean variable for showing plots all through the program.
    is_MMIA : bool
        Boolean variable for sepparating GLM expansion from MMIA expansion.

    Returns
    -------
    GLM_current_data : list
        List of daily lists of snippets like "GLM_int_Data" input, but with
        accomodation to 0.00001s timesteps done.
    '''

    # Expanding snippet to fit missing MMIA timesteps to cross-correlate data
    if is_MMIA == 0:
        print('Fitting GLM data in MMIA timesteps date %d snippet %d...' % (day,snippet))
    else:
        print('Completing MMIA data in MMIA timesteps date %d snippet %d...' % (day,snippet))

    new_length = 1          # New length of the timestep-wise matrix
    acumulated_voids = 0    # Number of non-existing timesteps up to current line
    void_info = np.zeros((len(GLM_int_data),4)) # Matrix of special info for each line:
```

```
        # 1st column: .txt row number
        # 2nd column: void timesteps after that row until next existing timestep
        # 3rd column: Accumulated void timesteps before current line
        # 4th column: Differential energy between existing timesteps ([i]-[i-1])

    # Updating new_length value to make a new table with 1st dimension being new_length

    GLM_int_data[0,0] = round(GLM_int_data[0,0],5)      # Rounding to MMIA period

    for j in range(1,len(GLM_int_data)):

        GLM_int_data[j,0] = round(GLM_int_data[j,0],5) # Rounding to MMIA period
        void_info[j][0] = j     # Filling first void_info column

        if GLM_int_data[j,0] == GLM_int_data[j-1,0] + 0.00001:  # Exactly one timestep ahead
            new_length = new_length + 1
            void_info[j][2] = acumulated_voids

        elif GLM_int_data[j,0] < GLM_int_data[j-1,0] + 0.00001: # Less than a whole timestep (sometimes occur)
            new_length = new_length + 1
            void_info[j][2] = acumulated_voids

        else:   # There are missing timesteps in between current and last row
            void_timesteps = round((GLM_int_data[j,0] - GLM_int_data[j-1,0])/0.00001) - 1
            new_length = new_length + 1 + void_timesteps
            void_info[j-1][1] = void_timesteps
            acumulated_voids = acumulated_voids + void_timesteps
            void_info[j][2] = acumulated_voids
            void_info[j-1][3] = GLM_int_data[j,1] - GLM_int_data[j-1,1]

    # Filling the new time-wise matrix

    GLM_current_data = np.zeros((new_length,2)) # New matrix with void lines for non-existing timesteps

    for j in range(0,len(GLM_int_data)):
        new_j = int(j + void_info[j,2])     # Row position in the new matrix
        GLM_current_data[new_j,:] = GLM_int_data[j,:] # Filling rows with existing data

        if void_info[j,1] != 0:    # Lines with non-existing timesteps afterwards
            counter = 1             # Adds 0.00001s and a linear energy fraction
```

```
            for k in range(new_j+1, new_j+1+int(void_info[j,1])):
                GLM_current_data[k,0] = GLM_int_data[j,0] + counter * 0.00001
                GLM_current_data[k,1] = GLM_int_data[j][1] + counter * (void_info[j][3]/void_info[j][1])
                counter = counter + 1

    if show_plots == 1 and is_MMIA == 0:
        # GLM time representation at MMIA sample rate
        plt.figure()
        plt.plot(GLM_current_data[:,0])
        plt.title('GLM Time vector of day %d snippet %d with 0.00001s period' % (day, snippet))
        plt.xlabel('Samples')
        plt.ylabel('Time [s]')
        plt.grid('on')
        plt.show()

        # Radiance vs time graph representation
        plt.figure()
        plt.plot(GLM_current_data[:,0],GLM_current_data[:,1], linewidth=0.5, color='black')
        plt.grid('on')
        plt.title('GLM signal of day %d snippet %d with MMIA sample rate (0.00001s)' % (day, snippet))
        plt.xlabel('Time (second of the day) [s]')
        plt.ylabel('Radiance [J]')
        plt.show()

    print('Date %d snippet %d fit' % (day, snippet))
    print(' ')

    return GLM_current_data
```

B.13 Function signal_delay.py

Listing B.13: Full code for signal_delay.py function

```
def signal_delay(data1, data2, show_plots, day, snip):
    '''
    This function determines the delay in samples of two signals using a
    cross-correlation method.

    Parameters
    ----------
    data1 : Array
        First signal to be analyzed. In this case, GLM signal.
    data2 : Array
        Second signal to be analyzed. In this case, MMIA signal.
    show_plots : bool
        Boolean variable for showing plots all through the program.
    day : int
```

```
        Day of the snip to be cross-correlated in shape YearMonthDay
    snip : int
        Position of snip in day "day"

    Returns
    -------
    real_delay_samples : The number of samples that data1 is shifted with
        respect to data2.
    '''

    xcorr_factors = correlate(data1[:,1], data2[:,1], mode='full', method = 'auto')

    len_x = len(data1)+len(data2)-1
    x = np.empty(len_x)

    for i in range(len_x):
        if (len_x % 2) == 0: # Even number
            x[i] = (i - (len_x/2))
        if (len_x % 2) != 0: # Odd number
            x[i] = (i - (len_x/2 - 0.5))

    if show_plots == 1:
        plt.subplot(2, 1, 1)
        plt.plot(data2[:,1],'-r', linewidth = 0.5)
        plt.plot(data1[:,1],'-k', linewidth = 0.5)
        #plt.title('Non-correlated GLM (black) and MMIA (red) signals, day %d snippet %d' % (day, snip))
        plt.ylabel('Normalized Energy')
        plt.xlabel('Vector samples')
        plt.legend(['MMIA signal', 'GLM signal'])
        plt.grid('on')

        plt.subplot(2, 1, 2)
        plt.plot(x, xcorr_factors, '-b', linewidth = 0.5)
        plt.xlabel('Diff. Samples')
        plt.ylabel('Correlation Factor')
        plt.grid('on')

    # Delay samples accounting actual positioning due to different lengths:
    max_factor_pos = np.where(xcorr_factors == max(xcorr_factors))[0][0]

    if ((len(data1)+len(data2)) % 2 == 0): # len(x) is Odd
        delay_samples = x[max_factor_pos]+(len(data1)-len(data2))/2

    if ((len(data1)+len(data2)) % 2 != 0): # len(x) is Even
        delay_samples = x[max_factor_pos]+(len(data1)-len(data2))/2 + 0.5

    # Delay samples accounting actual positioning due to time:
    if data1[0,0] > data2[0,0]: # GLM vector starts later
```

```
        pos_MMIA_start_GLM = np.where(data2[:,0] <= data1[0,0])
    [0][-1]
        real_delay_samples = delay_samples + pos_MMIA_start_GLM
    elif data1[0,0] < data2[0,0]: # GLM vector starts earlier
        pos_GLM_start_MMIA = np.where(data1[:,0] <= data2[0,0])
    [0][-1]
        real_delay_samples = delay_samples - pos_GLM_start_MMIA
    else:
        real_delay_samples = delay_samples

    return (real_delay_samples)
```

B.14 Function cross_correlate_GLM_MMIA.py

Listing B.14: Full code for cross_correlate_GLM_MMIA.py function

```
def cross_correlate_GLM_MMIA(GLM_snippets, MMIA_snippets, GLM_norm,
     MMIA_norm, matches, show_plots):
    '''
    This function gets snippets from GLM and MMIA and cross-
    correlates them
    to syncronize the signals and compare peaks.

    Parameters
    ----------
    GLM_snippets : list
        List of daily lists of GLM snippets.
    MMIA_snippets : list
        List of daily lists of MMIA snippets.
    GLM_norm : list
        List of daily lists of GLM normalized snippets.
    MMIA_norm : list
        List of daily lists of MMIA normalized snippets.
    matches : list
        List of existing GLM and MMIA dates.
    show_plots : bool
        Boolean variable for plotting. Not ploting makes the
    program faster.

    Returns
    -------
    GLM_xcorr : list
        List of daily lists of synchronized GLM data.
    MMIA_xcorr : list
        List of daily lists of synchronized MMIA data.
    delays : list
        List of daily lists of delay between GLM and MMIA signal
    per snippet.
    '''

    print('Starting cross-correlation of snippets and
    syncronization of signals...')

```

```
    # Creation of new lists for daily GLM and MMIA cross-correlated data
    GLM_xcorr = [None] * len(GLM_snippets)
    MMIA_xcorr = [None] * len(MMIA_snippets)
    delays = [None] * len(GLM_snippets)
    for i in range(len(GLM_snippets)):
        snips = [None] * len(GLM_snippets[i])
        GLM_xcorr[i] = snips
    for i in range(len(MMIA_snippets)):
        snips = [None] * len(MMIA_snippets[i])
        MMIA_xcorr[i] = snips
    for i in range(len(GLM_snippets)):
        snips = [None] * len(GLM_snippets[i])
        delays[i] = snips

    for i in range(len(GLM_snippets)):
        print('Date %d, %d / %d...' % (int(matches[i]), i+1, len(matches)))
        for j in range(len(GLM_snippets[i])):
            if i == 6 and j == 2:
                show_plots = 1
            else:
                show_plots = 0
            # If there's no info for this snippet due to lack of .nc or .cdf files

            if type(GLM_snippets[i][j]) == np.ndarray and type(MMIA_snippets[i][j]) == np.ndarray:

                current_GLM = GLM_norm[i][j]
                current_MMIA = MMIA_norm[i][j]

                if show_plots == 1:
                    # Plotting cross-correlated and syncronized GLM and MMIA normalized signals
                    plt.figure()
                    plt.plot(current_MMIA[:,0], current_MMIA[:,1], color = 'r', linewidth = 0.5)
                    plt.plot(current_GLM[:,0], current_GLM[:,1], color = 'black', linewidth = 1)
                    plt.legend(['MMIA','GLM'])
                    plt.title('GLM (black) and MMIA (red) correlated normalized signals for day %d snippet %d' % (int(matches[i]), j))
                    plt.xlabel('Time [s]')
                    plt.ylabel('Normalized energy')
                    plt.grid('on')
                    plt.show()

                # Calculation of delay in samples of GLM with respect to MMIA
                delay = int(TFG.signal_delay(current_GLM, current_MMIA, show_plots, int(matches[i]), j))

                delays[i][j] = delay

```

```
            MMIA_xc = current_MMIA
            GLM_xc = np.zeros((len(current_GLM),2))

            for k in range(len(current_GLM)):
                if delay != 0: # There is delay
                    # Adjust Normalized vector
                    GLM_xc[k,0] = current_GLM[k,0] - delay*0.00001
                    GLM_xc[k,1] = current_GLM[k,1]
                    # Adjust original vector
                    GLM_snippets[i][j][k,0] = GLM_snippets[i][j][k,0] - delay*0.00001
                else: # delay==0 so no delay at all
                    GLM_xc[k,:] = current_GLM[k,:]

            if show_plots == 1:
                # Plotting cross-correlated and syncronized GLM and MMIA normalized signals
                plt.figure()
                plt.plot(MMIA_xc[:,0], MMIA_xc[:,1], color = 'r', linewidth = 0.5)
                plt.plot(GLM_xc[:,0], GLM_xc[:,1], color = 'black', linewidth = 1)
                plt.legend(['MMIA','GLM'])
                plt.title('GLM (black) and MMIA (red) correlated normalized signals for day %d snippet %d' % (int(matches[i]), j))
                plt.xlabel('Time [s]')
                plt.ylabel('Normalized energy')
                plt.grid('on')
                plt.show()

                # Plotting cross-correlated and syncronized GLM and MMIA non normalized signals
                plt.figure()
                plt.plot(MMIA_snippets[i][j][:,0], MMIA_snippets[i][j][:,1], color = 'r', linewidth = 0.5)
                plt.plot(GLM_snippets[i][j][:,0], GLM_snippets[i][j][:,1], color = 'black', linewidth = 1)
                plt.legend(['MMIA','GLM'])
                plt.title('GLM (black) and MMIA (red) correlated normalized signals for day %d snippet %d' % (int(matches[i]), j))
                plt.xlabel('Time [s]')
                plt.ylabel('Energy')
                plt.grid('on')
                plt.show()

            GLM_xcorr[i][j] = GLM_snippets[i][j]
            MMIA_xcorr[i][j] = MMIA_snippets[i][j]

            print('Date %d snippet %d cross-correlated and aligned!' % (int(matches[i]), j))
        elif type(GLM_snippets[i][j]) == np.ndarray and type(MMIA_snippets[i][j]) != np.ndarray:
```

```
                print('Date %d snippet %d was pre-avoided for lack of MMIA data' % (int(matches[i]), j))
            elif type(GLM_snippets[i][j]) != np.ndarray and type(MMIA_snippets[i][j]) == np.ndarray:
                print('Date %d snippet %d was pre-avoided for lack of GLM data' % (int(matches[i]), j))
            else:
                print('Date %d snippet %d was pre-avoided for lack of GLM and MMIA data' % (int(matches[i]), j))

        print(' ')
    print('All snippets checked!')
    print(' ')
    return [GLM_xcorr, MMIA_xcorr, delays]
```

B.15 Function get_GLM_MMIA_peaks.py

LISTING B.15: Full code for get_GLM_MMIA_peaks.py function

```
def get_GLM_MMIA_peaks(GLM_xcorr, MMIA_xcorr, matches, show_plots):
    '''
    This function gets the cross-correlated vector snippets from GLM and MMIA
    and finds their indexes for every prominent peak in their signals.
    It returns a list of lists of index vectors for every snippet.

    Parameters
    ----------
    GLM_xcorr : list
        List of daily lists of synchronized GLM data.
    MMIA_xcorr : list
        List of daily lists of synchronized MMIA data.
    matches : list
        List of existing GLM and MMIA dates
    show_plots : bool
        Boolean variable for plotting. Not ploting makes the program faster.

    Returns
    -------
    GLM_peaks : list
        List of daily lists with vectors of GLM_xcorr indexes for peaks in the
        signal.
    GLM_peaks : list
        List of daily lists with vectors of MMIA_xcorr indexes for peaks in the
        signal.
    '''

    print('Finding peaks in GLM and MMIA cross-correlated signals')
    print('This process can take a while...')
```

```
    GLM_peaks = [None] * len(GLM_xcorr)
    MMIA_peaks = [None] * len(MMIA_xcorr)
    for i in range(len(GLM_xcorr)):
        snips = [None] * len(GLM_xcorr[i])
        GLM_peaks[i] = snips
    for i in range(len(MMIA_xcorr)):
        snips = [None] * len(MMIA_xcorr[i])
        MMIA_peaks[i] = snips

    for i in range(len(GLM_xcorr)):
        print('Date %s, %d / %d...' % (matches[i], i+1, len(GLM_xcorr)))
        for j in range(len(GLM_xcorr[i])):

            if type(GLM_xcorr[i][j]) == np.ndarray and type(MMIA_xcorr[i][j]) == np.ndarray:

                print('Finding peaks for date %s snippet %d / %d' % (matches[i], j, len(GLM_xcorr[i])))

                # Cropping in order to have the same time to compare

                # Not overlapping conditions
                GLM_left_cond = GLM_xcorr[i][j][-1,0]<=MMIA_xcorr[i][j][0,0]
                GLM_right_cond = GLM_xcorr[i][j][0,0]>=MMIA_xcorr[i][j][-1,0]

                if GLM_left_cond == True or GLM_right_cond == True:
                    print('Correlated snippets for date %s snippet %d do not overlap at all' % (matches[i], j))
                else:

                    # Finding the starting position
                    GLM_first = 0
                    if GLM_xcorr[i][j][0,0] < MMIA_xcorr[i][j][0,0]: # GLM starts first
                        start_pos = np.where(GLM_xcorr[i][j][:,0] <= MMIA_xcorr[i][j][0,0])[0][-1]
                        GLM_first = 1
                    elif GLM_xcorr[i][j][0,0] > MMIA_xcorr[i][j][0,0]: # MMIA starts first
                        start_pos = np.where(MMIA_xcorr[i][j][:,0] <= GLM_xcorr[i][j][0,0])[0][-1]
                    else: # Both start at the sime timestep
                        start_pos = 0

                    # Finding the end position
                    GLM_last = 0
                    if GLM_xcorr[i][j][-1,0] < MMIA_xcorr[i][j][-1,0]: # GLM ends first
                        end_pos = np.where(MMIA_xcorr[i][j][:,0] <= GLM_xcorr[i][j][-1,0])[0][-1]
                    elif GLM_xcorr[i][j][-1,0] > MMIA_xcorr[i][j][-1,0]: # MMIA ends first
```

```
                        end_pos = np.where(GLM_xcorr[i][j][:,0] <= MMIA_xcorr[i][j][-1,0])[0][-1]
                        GLM_last = 1
                    else: # Both end at the sime timestep
                        end_pos = -1

                    # Cropping vectors accordingly
                    if GLM_first == 1 and GLM_last == 1:
                        GLM_vector = GLM_xcorr[i][j][start_pos:end_pos,1]
                        GLM_time_vector = GLM_xcorr[i][j][start_pos:end_pos,0]
                        MMIA_vector = MMIA_xcorr[i][j][:,1]
                        MMIA_time_vector = MMIA_xcorr[i][j][:,0]

                    elif GLM_first == 1 and GLM_last != 1:
                        GLM_vector = GLM_xcorr[i][j][start_pos:-1,1]
                        GLM_time_vector = GLM_xcorr[i][j][start_pos:-1,0]
                        MMIA_vector = MMIA_xcorr[i][j][0:end_pos,1]
                        MMIA_time_vector = MMIA_xcorr[i][j][0:end_pos,0]

                    elif GLM_first != 1 and GLM_last == 1:
                        GLM_vector = GLM_xcorr[i][j][0:end_pos,1]
                        GLM_time_vector = GLM_xcorr[i][j][0:end_pos,0]
                        MMIA_vector = MMIA_xcorr[i][j][start_pos:-1,1]
                        MMIA_time_vector = MMIA_xcorr[i][j][start_pos:-1,0]

                    elif GLM_first != 1 and GLM_last != 1:
                        GLM_vector = GLM_xcorr[i][j][:,1]
                        GLM_time_vector = GLM_xcorr[i][j][:,0]
                        MMIA_vector = MMIA_xcorr[i][j][start_pos:end_pos,1]
                        MMIA_time_vector = MMIA_xcorr[i][j][start_pos:end_pos,0]


                    # Calculating indexes of peaks in GLM signal
                    GLM_peak_vec, _ = find_peaks(GLM_vector, prominence = 0.3e-14, rel_height = 20)

                    # Calculating indexes of peaks in MMIA signal
                    MMIA_noise_level = np.percentile(MMIA_vector,90, axis=0)
                    MMIA_peak_vec, _ = find_peaks(MMIA_vector, rel_height = 100, height = MMIA_noise_level, prominence = 0.4, distance=400)

                    GLM_peaks[i][j] = GLM_peak_vec
                    MMIA_peaks[i][j] = MMIA_peak_vec

```

```python
                    # Cropping GLM_xcorr and MMIA x_corr
                    GLM_xcorr_new_snippet = np.zeros((len(GLM_vector),2))
                    GLM_xcorr_new_snippet[:,0] = GLM_time_vector
                    GLM_xcorr_new_snippet[:,1] = GLM_vector
                    GLM_xcorr[i][j] = GLM_xcorr_new_snippet

                    MMIA_xcorr_new_snippet = np.zeros((len(MMIA_vector),2))
                    MMIA_xcorr_new_snippet[:,0] = MMIA_time_vector
                    MMIA_xcorr_new_snippet[:,1] = MMIA_vector
                    MMIA_xcorr[i][j] = MMIA_xcorr_new_snippet

                    if show_plots == 1:

                        plt.figure()
                        plt.plot(GLM_vector, color = 'black', linewidth=0.5)
                        plt.plot(GLM_peak_vec, GLM_vector[GLM_peak_vec], "*", color='b')
                        plt.title('GLM peaks on day %d, snippet %d' % (int(matches[i]), j))
                        plt.xlabel('Samples')
                        plt.ylabel('Radiance [J]')
                        plt.grid('on')
                        plt.show()

                        plt.figure()
                        plt.plot(MMIA_vector, color = 'r', linewidth=0.5)
                        plt.plot(MMIA_peak_vec, MMIA_vector[MMIA_peak_vec], "*", color='b')
                        plt.title('MMIA peaks on day %d, snippet %d' % (int(matches[i]), j))
                        plt.xlabel('Samples')
                        plt.ylabel(r'Irradiance $\left[\dfrac{\mu W}{m^2}\right]$')
                        plt.grid('on')
                        plt.show()

            else:
                print('Date %s snippet %d was not cross correlated' % (matches[i], j))
        print(' ')

    print('Done!')
    return [GLM_peaks, MMIA_peaks]
```