Lightcurves Documentation

Adi Raj

Contents

1 Introduction

This Lightcurves repository provides an end-to-end solution for discovering, analyzing, and visualizing astronomical light curves—time-series brightness variations of celestial objects. Light curves are critical in astrophysics for uncovering details about object variability, classification, and potential underlying astrophysical processes such as accretion or flaring events.

Central to this project is the use of the *Chandra Source Catalog* (CSC), which contains extensive observations from the *Chandra X-ray Observatory*. By leveraging *Chandra*'s exceptional angular resolution and sensitivity, researchers can identify X-ray sources and extract light curves that offer insight into source variability over time. The ability to retrieve data from the CSC, filter it for significance or counts thresholds, and then produce light curves—alongside smaller "postage stamp" images for spatial context—can be especially valuable in multi-wavelength studies, transient tracking, or for catalogs of active galactic nuclei (AGN), X-ray binaries, and other variable X-ray sources.

This documentation covers not only the high-level functionalities but also the design considerations that enable flexible expansion and future improvements. Whether your focus is on investigating particular sources, testing algorithms for transient detection, or simply exploring new data analysis pipelines, this project aims to streamline common tasks. By handling data retrieval, quality checks, automated plotting, and dynamic re-binning of data, it reduces the overhead of repeated tasks and frees time for deeper scientific inquiry.

2 Project Overview

This project is designed around a modular and adaptable framework to support various astronomical data-processing workflows, particularly those involving X-ray observations. Each of the core modules focuses on a specific stage of the data journey: from finding relevant sources in the CSC, downloading and preparing the raw data, creating and refining light curves, all the way to generating visual outputs. This project focuses on leveraging high-sensitivity X-ray observations from *Chandra*, *XMM-Newton*, and other space-based observatories to systematically identify and analyze short-time-scale variability in extragalactic X-ray binaries (XRBs). By reprocessing data with specialized software—such as CIAO for *Chandra* and the Science Analysis System (SAS) for *XMM-Newton*—the primary goal is to generate background-subtracted, uniformly calibrated light curves. These time-series datasets capture fluctuations in X-ray intensity that could signal astrophysical transits or eclipses, potentially caused by orbiting objects like exoplanets. The project further relies on a wide suite of software tools, including FTOOLS tasks within HEASoft (e.g., *lcurve*, *lcmath*, and *fxbary*), to filter, model, and scrutinize each light curve for signatures of dips or eclipses. By applying techniques from machine learning and classical time-series analysis, we aim to pinpoint and characterize transient phenomena—such as X-ray flares or periodic intensity dips—across thousands of potential targets.

3 Cloning and Using the Program

To get started with the program, ensure you have Git and Python (version 3.7 or higher) installed on your system. Also note that CIAO 4.15 must be installed on your system in addition to the other project dependencies. Follow these steps:

1. Open your terminal and run the following command:

```
$ git clone https://github.com/SenatorTreason05/Lightcurves.git
```

   This command creates a local copy of the repository in a folder named `Lightcurves`.

2. Change into the repository directory by running:

```
$ cd Lightcurves
```

3. It is best to use a conda environment or virtual environment to manage dependencies. For example, if you are using conda, create and activate an environment as follows:

```
$ conda create -n lightcurves_env python=3.8
$ conda activate lightcurves_env
```

   Then install all required Python packages with:

```
$ pip install -r requirements.txt
```

   Of course, ensure that CIAO 4.15 is installed and properly configured on your system before running the program.

4. In the correct conda environment, you can launch the main pipeline with either of the following commands:

```
$ python main.py
```

   or, to run without the GUI,

```
$ python main.py --no-gui
```

   This command clears the terminal, parses command-line arguments, searches for sources in the Chandra Source Catalog, downloads the data, and processes the retrieved sources. Optionally, if configured, it may also launch the web server to display results.

5. To start the server for interactive manipulation of the results, run:

```
$ python server.py
```

   When prompted, provide the absolute path to the index file. For example:

```
/Users/mihirpatankar/Documents/Projects/Lightcurves/output/M3
    -2023-12-08_18:07:31/index.html
```

   This will launch a web interface where you can view and interact with your processed data.

6. You can also run the tool on a batch of objects. First, list your objects in a file located at `batch_run/objects_list.txt` (one object per line). Then, execute:

```
$ python batch_run/batch_run.py
```

   The progress of the batch run will be continuously output to `batch_run/current_progress.txt`.

7. Lastly, to update your local copy with any new changes from the repository, run:

```
$ git pull
```

4 File Structure

This section serves as a basic outline for each file that outlines how they interact.

data_structures.py
Defines named tuples and other data constructs shared across modules.

- `ConfigField` represents a GUI configuration entry, facilitating the display and validation of user input in the `search_config.py` module.
- `Message` holds string content and an optional UUID, allowing for tracked logging or message passing within multiprocessing queues.
- `DataProducts` collects references to core observation files (e.g., event lists, region files) that are essential for generating and analyzing light curves.
- `CountsChecker` integrates a queue and cancel event to manage real-time count verification tasks across processes.
- By housing these definitions in a dedicated file, the project centralizes and standardizes data modeling. This makes code throughout the pipeline more readable and reduces the potential for naming inconsistencies.

exporter.py
Manages the collation and writing of processed data to disk, particularly HTML outputs.

- The `add_source()` method accepts results generated for a specific source and stores them in a master data structure for final export.
- `write_plot_image()` takes raw SVG plot output, writes it to disk, and optionally saves an associated CSV. This method ensures that visual artifacts of the analysis (e.g., light curve plots) remain well-organized.
- `export()` merges all processed information into a single HTML page via a Jinja2 template (`output_template.jinja`), enabling quick sharing or interactive viewing of the results.
- By decoupling export logic from processing logic, the system keeps data generation code clean and flexible. The `exporter.py` module can also be extended to include other file formats (PDF, JSON) if needed.

lightcurve_generator.py
Oversees creating light curves from observation data, including binning and preliminary filtering.

- Responsible for orchestrating the end-to-end generation of light curves, starting from raw event files. Centralizes logic such as time-binning and data consolidation.
- `dispatch_source_processing()` coordinates processing for multiple sources in a queue-based, possibly multi-threaded environment. Each source is then passed on to `process_source()`.
- `generate_lightcurve()` directs how data is binned (via `apply_binning()`) and calls `export_lightcurve()` to store the final CSV data.

- Integrates easily with CIAO, *XMM-Newton* SAS outputs, or other event-based data formats. If future missions require different event file structures, additional adapters can be added without disrupting the project's overall design.

lightcurve_processing.py
Applies post-processing techniques to generated light curves, including noise filtering, normalization, and anomaly detection.

- The `LightcurveProcessor` class includes functions to load raw CSV data (`load_data()`), filter out noise or unwanted data points (`apply_filtering()`), and normalize counts (`normalize_lightcurve()`).
- `export_processed_lightcurve()` can write a final, refined light curve to disk for advanced scientific analysis or direct ingestion by machine learning pipelines.
- By maintaining a separate module for post-processing, the project can be easily extended to include advanced algorithms such as wavelet transforms, Lomb–Scargle periodograms, or custom ML-based outlier detection.

postage_stamp_plotter.py
Produces "postage stamp" images from FITS files, offering quick visual snapshots around particular X-ray sources.

- `PostageStampPlotter` class focuses on reading images via `load_image()`, selecting a region of interest with `crop_region()`, optionally applying log scaling, and finally saving a small, labeled image.
- Particularly useful for verifying source detections or investigating extended emission around an object.
- Exports to convenient formats (e.g., PNG) for direct embedding in HTML reports or for visual inspection in DS9 or other imaging tools.

search.py
Handles query operations against the Chandra Source Catalog and orchestrates data downloads for discovered sources.

- `SourceManager` is the core class that runs the search against CSC, filters results by significance, and initiates the download of event files.
- Contains `search_csc()`, which uses `astropy` coordinates and catalogs to conduct a cone search around a user-specified coordinate or object name.
- `download_data_products()` retrieves relevant files from the archive (optionally in parallel). These files form the foundational input for subsequent modules.
- Also includes a `Terminal` class for clearing the screen and writing progress updates in a neat, multi-column format—ensuring the user has real-time feedback during long downloads.

search_config.py
Manages configuration settings and provides an optional GUI for user adjustments.

- `SearchConfigGUI` allows the user to load, modify, and save configuration parameters (e.g., search radius, binsize) from a YAML file.
- `load_config()` attempts to parse the file into Python objects, while `save_config()` writes updated values back to disk.
- The separation of config logic ensures that global parameters—like significance thresholds or default radial search boundaries—remain consistent throughout the pipeline without "hard coding" them in multiple places.

server.py
Implements a lightweight Flask-based server to deliver processed HTML and support dynamic re-binning or recalculation requests.

- `start()` spins up a Flask application that serves the main HTML report (often generated by `exporter.py`).
- Defines routes like `index()` for rendering the main page and `serve_image()` to provide static files (e.g., PNG or SVG plots) to the browser.
- A `recalculate()` endpoint demonstrates how to implement real-time user interaction, allowing the user to specify a new time bin size or other parameter. The server then triggers partial reprocessing of light curves and updates the displayed plots without stopping the application.
- This live interactive mode greatly enhances the user experience for large data sets, enabling immediate feedback and experimentation.

output_template.jinja
A Jinja2 template used by `exporter.py` (and optionally by `server.py`) to render final HTML reports.

- Structures the page layout with placeholders for source names, observation IDs, light curve plots, and descriptive statistics.
- Contains logic for iterating over sources and automatically generating nested HTML tables or lists. This is where interactive elements—like re-binning input fields—can be seamlessly integrated with JavaScript.
- By storing the template separately, the project cleanly separates the data processing code from the user-facing presentation logic.

5 Detailed File Descriptions

5.1 main.py

This file starts the program and manages the overall workflow, from searching for data to processing and displaying results. It serves as the central launch point for the project.

Technical Overview: `main.py` serves as the project's primary entry point and is where the entire source-searching and light-curve-generation pipeline begins. The core function, `main()`, ensures that each major step in the workflow is executed in the correct order while offering a flexible interface (GUI or command-line only) for user interaction. Below is an overview of the streamlined process:

1. Clears the terminal output using the command `subprocess.run(["clear"], check=False)`.
2. Parses command-line arguments to determine if the GUI should be used.
3. Initializes the `SourceManager` class from `search.py` with the appropriate configuration settings.
4. Calls `search_csc()` to search for sources in the Chandra Source Catalog.
5. Calls `download_and_process()` to download data and process the retrieved sources.
6. If the configuration setting `"Auto Start Server"` is enabled, the function starts the web server to display results.

```
def main():
    """Entry point for the program."""
    subprocess.run(["clear"], check=False)  # Clears terminal output

    # Parse command-line arguments
    argument_parser = argparse.ArgumentParser()
    argument_parser.add_argument("--no-gui", action="store_false")
    arguments = argument_parser.parse_args()

    # Initialize SourceManager with configuration
    source_manager = search.SourceManager(search_config.get_config(use_gui=
        arguments.no_gui))

    # Perform source search
    source_manager.search_csc()

    # Download and process sources
    output_html_file = source_manager.download_and_process()

    # Start server if enabled in configuration
    if source_manager.config["Auto Start Server"]:
        server.start(output_html_file)
```

5.2 data_structures.py

This module defines the basic data formats used throughout the project so that all parts of the program understand the information in the same way. It ensures consistency in how data is stored and passed between components.

Technical Overview: The `data_structures.py` module plays a crucial role in the project by defining the standardized data representations that are used throughout various components of the pipeline. By encapsulating key data elements as immutable structures, this module ensures consistency, type-safety, and ease of access across the application.

Classes:

1. `ConfigField`: Represents a GUI configuration field.

```
class ConfigField(NamedTuple):
    """Holds immutable data for a GUI configuration entry."""
    label: Label
    field: Entry | Checkbutton
    default_value: str | bool
    entry_type: Callable
```

2. `Message`: Encapsulates a message with an optional UUID.

```
class Message(NamedTuple):
    """Encapsulates a message with an optional UUID."""
    content: str
    uuid: UUID = None
```

3. `DataProducts`: Holds paths to relevant observation files.

```
class DataProducts(NamedTuple):
    """Holds paths to observation files used in the processing pipeline
        ."""
    event_list_file: str
    source_region_file: str
```

4. `CountsChecker`: Manages count verification queues.

```
class CountsChecker(NamedTuple):
    """Tracks count verification in observation processing."""
    queue: multiprocessing.Queue
    cancel_event: multiprocessing.Event
```

5.3 exporter.py

Exporter.py collects the processed results and formats them into a final report. It takes data from various parts of the pipeline and outputs it in a clear, organized form.

Technical Overview: `exporter.py` is the module responsible for gathering processed data from various stages of the pipeline – such as light curve generation and processing – and formatting it into a cohesive, accessible output. It provides functions that collate results from modules such as `search.py`, `lightcurve_generator.py`, and `lightcurve_processing.py` into a master data structure. This consolidated data is then rendered into an HTML report using a Jinja2 template (from `output_template.jinja`), which can be served via the Flask web server implemented in `server.py`. In this way, `exporter.py` acts as the final bridge between raw processed outputs and the user-facing visualization, ensuring that the comprehensive results of the data analysis pipeline are presented in an organized and user-friendly format.

Functions:

- `add_source()`: Aggregates data for sources and stores plots.
- `write_plot_image()`: Saves light curve plots.
- `export()`: Writes output to an HTML file.

`add_source(source_name, results)` This function collects processed data for a given source and prepares it for export.

```
def add_source(self, source_name, results):
    """Aggregates processed results for a given source."""
    self.master_data[source_name] = results
```

`write_plot_image(plot_directory, observation_id, plot_svg_data, plot_csv_data)` This function saves light curve plots to disk in both SVG and CSV formats.

```
def write_plot_image(self, plot_directory, observation_id, plot_svg_data,
    plot_csv_data):
    """Saves the generated light curve plots as images."""
    plot_path = plot_directory / f"{observation_id}.svg"
    with open(plot_path, "w", encoding="utf-8") as plot_file:
        plot_file.write(plot_svg_data)
    return plot_path
```

`export()` This function writes all the processed data into an HTML file for visualization.

```
def export(self):
    """Writes processed data to an HTML output file."""
    with open(output := self.output_directory / "index.html", mode="w",
        encoding="utf-8") as file:
        environment = Environment(loader=FileSystemLoader("./"))
        template = environment.get_template("Lightcurves/output_template.
            jinja")
        content = template.render(source_data=self.master_data)
        file.write(content)
```

5.4 server.py

Server.py runs a simple web server to display the final outputs and reports. It makes the results accessible through a web interface for easier review.

Technical Overview: `server.py` implements a lightweight Flask web server that delivers the final HTML output generated by `exporter.py` and serves static assets such as plots and images. It provides an interactive interface for users to view and manipulate the results of the data processing pipeline. This ties together outputs from modules like `search.py` and `lightcurve_processing.py`, thus enabling dynamic exploration and real-time reprocessing of the analyzed data.

Functions:

- `start()`: Starts the Flask web server and sets up routes.
- `index()`: Serves the main HTML page with processed data.
- `serve_image()`: Serves images for visualization.
- `recalculate()`: Handles requests to reprocess and rebin data.

`start(html_index_file)` This function initializes and starts the Flask web server, loading processed HTML results and serving them dynamically.

```
def start(html_index_file: Path):
    """Start the web server and set up routes."""
    app = flask.Flask(__name__)
    status_message_queue = Queue()
    app.run(debug=True, use_reloader=False)
```

`index()` Serves the main processed HTML page.

```
@app.route("/", methods=["GET"])
def index():
    with open(html_index_file, mode="r", encoding="utf-8") as file:
        return flask.render_template_string(file.read())
```

`serve_image(filename)` Serves static image files used in data visualization.

```
@app.route("/<path:filename>", methods=["GET"])
def serve_image(filename):
    file_path = Path(filename)
    return flask.send_from_directory(html_index_file.parent, file_path)
```

`recalculate()` Handles requests to reprocess and rebin observation data.

```
@app.route("/recalculate", methods=["POST"])
def recalculate():
    config = search_config.get_config()
    request_body = flask.request.get_json()
    source_name = request_body["sourceName"]
    observation_id = request_body["observationID"]
    new_binsize = float(request_body["newBinsize"])

    status_message_queue.put("Reprocessing Observation...")
    processing_results = reprocess_source(source_name, observation_id,
        new_binsize)
    if not processing_results:
        status_message_queue.put("Invalid Binsize!")
        flask.abort(400)
    return {"newPlotPath": str(processing_results.plot_svg_data)}
```

5.5 search.py

Search.py handles the task of looking up and downloading data from the Chandra Source Catalog. It gathers the raw observational data that the rest of the pipeline will process.

Technical Overview: `search.py` is the module that manages the discovery and retrieval of observational data. It queries the Chandra Source Catalog based on user-defined criteria, filters the results for significance, and initiates the download of necessary data products. By coordinating these tasks, `search.py` supplies the raw inputs that later modules – such as `lightcurve_generator.py` and `lightcurve_processing.py` – require for further analysis, making it an essential starting point of the data pipeline.

Classes:

- `SourceManager`: Handles searching, downloading, and processing sources.
- `Terminal`: Manages terminal output and progress display.

5.5.1 SourceManager Class

This class orchestrates the source searching, downloading, and processing workflow.

`search_csc()` Searches for sources in the Chandra Source Catalog using the provided criteria.

```
def search_csc(self):
    """Queries the Chandra Source Catalog for sources matching search
        criteria."""
    search_radius = self.config["Search Radius (arcmin)"] * units.arcmin
    object_name = self.config["Object Name"]
    try:
        sky_coord = SkyCoord.from_name(object_name)
        search_results = dal.conesearch(
            "http://cda.cfa.harvard.edu/csc2scs/coneSearch",
            sky_coord, search_radius, verbosity=2)
    except NameResolveError:
        print(f'No results for object "{object_name}".')
        sys.exit(1)
    print(f"Found {len(search_results)} sources.")
    self.sources = search_results
```

`filter_significant_sources()` Filters search results based on a significance threshold.

```
def filter_significant_sources(self):
    """Filters sources based on a defined significance threshold."""
    significance_threshold = self.config["Significance Threshold"]
    self.sources = [s for s in self.sources if s["significance"] >=
        significance_threshold]
    print(f"Filtered sources to {len(self.sources)} significant sources.")
```

`download_data_products()` Downloads relevant data products for a given source from CSC.

```
def download_data_products(download_directory, right_ascension, declination
    ):
    """Downloads data products using CIAO tools."""
    search_csc(
        pos=f"{right_ascension}, {declination}",
        radius="1.0",
        outfile="search-csc-outfile.tsv",
        download="all",
        root=download_directory)
```

`download_all_data_products()` Manages multi-threaded downloading of data products.

```
def download_all_data_products(self):
    """Iterates through CSC search results and downloads their data
       products."""
    for source in self.sources:
        source_directory = Path(self.config["Data Directory"]) / source["
            name"]
        self.download_data_products(source_directory, source["ra"], source[
            "dec"])
```

`process()` Handles processing of downloaded sources.

```
def process(self, process_error_event):
    """Processes downloaded sources using LightcurveGenerator."""
    lightcurve_generator = LightcurveGenerator(self.config)
    try:
        lightcurve_generator.dispatch_source_processing(self.
            downloaded_source_queue)
    except Exception:
        process_error_event.set()
        raise
```

5.5.2 Terminal Class

This class manages terminal output and progress visualization.

`clear()` Clears the terminal screen.

```
@staticmethod
def clear():
    """Clears the terminal output using ANSI codes."""
    print("\033[H\033[J", end="")
```

`write_columns()` Formats and writes messages in a two-column layout.

```
@classmethod
def write_columns(cls, left_column, right_column):
    """Displays messages in a structured two-column format."""
    column_width = cls.width() // 2
```

```
    for left_msg, right_msg in zip_longest(left_column, right_column,
        fillvalue=""):
        print(f"{left_msg.ljust(column_width)}{right_msg.ljust(column_width
            )}")
```

5.6 search_config.py

This module manages the settings for the project, loading and saving user-defined parameters from a configuration file. It allows users to adjust search criteria and processing options easily.

Technical Overview: `search_config.py` centralizes the configuration management for the entire pipeline by loading, validating, and saving user-defined settings (such as search criteria, data paths, and processing parameters) from a YAML file. This ensures consistent parameter usage across modules like `search.py`, `lightcurve_generator.py`, and `server.py`, and optionally provides a GUI interface for easy parameter adjustment.

Classes:

- `SearchConfigGUI`: Provides a graphical interface for modifying configuration settings.

5.6.1 SearchConfigGUI Class

This class provides a graphical interface for editing configuration settings. It loads configuration parameters, allows users to modify them, and saves the changes.

`__init__()` Initializes the configuration GUI with default settings.

```
def __init__(self, config):
    """Initializes the GUI for editing program configuration."""
    self.config = config
    self.config_entries = []
```

`populate_form_fields()` Loads existing configuration values into the GUI form fields.

```
def populate_form_fields(self):
    """Fills the form with stored or default configuration values."""
    for row, config_entry in enumerate(self.config_entries):
        input_label, input_field = (config_entry.label, config_entry.field)
        input_label.grid(row=row, column=0)
        input_field.grid(row=row, column=1)
        label_text = input_label["text"]
        self.change_input_value(
            input_field,
            self.config[label_text] if label_text in self.config else
                config_entry.default_value,
        )
```

`reset_form()` Resets all input fields to their default values.

```
def reset_form(self):
    """Resets all form fields to default values."""
    for config_entry in self.config_entries:
        self.change_input_value(config_entry.field, config_entry.
            default_value)
```

`finalize_config()` Validates and saves the user-edited configuration.

```
def finalize_config(self):
    """Extracts user inputs and returns the updated configuration
       dictionary."""
    def get_value(field):
        is_checkbox = isinstance(field, Checkbutton)
        return field.instate(["selected"]) if is_checkbox else field.get()

    try:
        validated_config = {
            config_entry.label["text"]: config_entry.entry_type(get_value(
                config_entry.field))
            for config_entry in self.config_entries
        }
    except ValueError as error:
        print(f"Invalid input - {error}.\nConfig will not be overwritten.")
        sys.exit(1)
    return validated_config
```

`save_config()` Saves the updated configuration to a YAML file.

```
def save_config(self, config):
    """Writes the configuration settings to a YAML file."""
    with open(CONFIG_FILE_PATH, "w", encoding="utf-8") as file:
        yaml.dump(config, file)
```

`load_config()` Loads configuration settings from a YAML file.

```
def load_config():
    """Loads configuration from the YAML file or returns default settings.
       """
    try:
        with open(CONFIG_FILE_PATH, "r", encoding="utf-8") as file:
            return yaml.safe_load(file)
    except (FileNotFoundError, ScannerError):
        print("Error loading config file, using default settings.")
        return {}
```

5.7 lightcurve_generator.py

This module converts raw observational data into light curves by organizing and binning the data over time. It lays the groundwork for further analysis by creating the initial time-series data.

Technical Overview: `lightcurve_generator.py` is the core module responsible for transforming raw event data into structured light curves. It performs critical tasks such as time binning and initial data filtering, creating the foundational time-series data required for further analysis. The processed light curves generated here are then utilized by modules like `lightcurve_processing.py` for advanced filtering and normalization, and by `exporter.py` for visualization.

Classes:

- `LightcurveGenerator`: The core class responsible for orchestrating the light curve generation process.

5.7.1 LightcurveGenerator Class

This class oversees the light curve extraction and processing workflow. It handles the collection of data products, applies filtering and binning operations, and ensures data is exported correctly.

`__init__()` Initializes the light curve generator with the configuration settings.

```
def __init__(self, config):
    """Initializes LightcurveGenerator with the given configuration."""
    self.config = config
    self.process_queue = Queue()
```

`dispatch_source_processing()` Manages the parallel processing of multiple sources.

```
def dispatch_source_processing(self, downloaded_source_queue):
    """Handles processing for all downloaded sources in a multi-threaded
       manner."""
    while True:
        source_directory = downloaded_source_queue.get()
        if source_directory is None:
            break
        self.process_source(source_directory)
```

`process_source()` Processes a single source directory and extracts light curves.

```
def process_source(self, source_directory):
    """Processes a given source directory and generates a light curve."""
    event_file = source_directory / "event.lc"
    if not event_file.exists():
        print(f"No event file found in {source_directory}")
        return
    self.generate_lightcurve(event_file)
```

`generate_lightcurve()` Applies binning and filtering to generate a light curve from raw event data.

```
def generate_lightcurve(self, event_file):
    """Generates a light curve from an event file using binning techniques.
       """
    binsize = self.config["Binsize"]
    lightcurve_data = self.apply_binning(event_file, binsize)
    self.export_lightcurve(lightcurve_data, event_file.parent)
```

`apply_binning()` Bins the data using a specified time interval.

```
def apply_binning(self, event_file, binsize):
    """Applies time binning to the event data to create a light curve."""
    data = self.load_event_data(event_file)
    binned_data = bin_data(data, binsize)
    return binned_data
```

`export_lightcurve()` Exports the processed light curve data.

```
def export_lightcurve(self, lightcurve_data, output_directory):
    """Saves the processed light curve to the output directory."""
    output_file = output_directory / "lightcurve.csv"
    lightcurve_data.to_csv(output_file)
    print(f"Exported light curve to {output_file}")
```

5.8 lightcurve_processing.py

Lightcurve_processing.py cleans and refines the raw light curves by filtering noise and normalizing values. It prepares the data for deeper analysis and better visualization.

Technical Overview: `lightcurve_processing.py` refines the raw light curves by applying statistical filtering, noise reduction, and normalization. This module prepares the data for in-depth analysis and visualization, ensuring that subsequent outputs are accurate and meaningful before being passed on to modules like `exporter.py` for final reporting.

Classes:

- `LightcurveProcessor`: Handles the main pipeline for processing light curve data.

5.8.1 LightcurveProcessor Class

This class manages all aspects of light curve processing, including loading raw data, applying statistical filtering, and exporting final results.

`__init__()` Initializes the processor with the required configuration.

```
def __init__(self, config):
    """Initializes the LightcurveProcessor with configuration settings."""
    self.config = config
```

`load_data()` Loads the raw light curve data from an event file.

```
def load_data(self, event_file):
    """Loads raw light curve data from an event file."""
    data = pd.read_csv(event_file)
    return data
```

`apply_filtering()` Applies statistical noise reduction and removes outliers.

```
def apply_filtering(self, data):
    """Applies noise filtering techniques to clean the light curve data."""
    filtered_data = data[data["counts"] > self.config["Min Counts"]]
    return filtered_data
```

`normalize_lightcurve()` Normalizes the light curve for further analysis.

```
def normalize_lightcurve(self, data):
    """Normalizes the light curve data by subtracting the mean and scaling.
       """
    data["counts"] = (data["counts"] - data["counts"].mean()) / data["
       counts"].std()
    return data
```

`export_processed_lightcurve()` Exports the processed and filtered light curve data to a CSV file.

```
def export_processed_lightcurve(self, data, output_directory):
    """Saves the processed light curve to a CSV file."""
    output_file = output_directory / "processed_lightcurve.csv"
    data.to_csv(output_file, index=False)
    print(f"Processed light curve exported to {output_file}")
```

5.9 postage_stamp_plotter.py

This file creates small, focused images from larger astronomical images, highlighting specific areas around X-ray sources. It generates visual snapshots that provide spatial context for the observations.

Technical Overview: `postage_stamp_plotter.py` generates compact visual snapshots – "postage stamp" plots – from FITS image files. It loads full-scale astronomical images, extracts focused regions around specific X-ray sources, applies appropriate scaling to enhance detail, and then exports these snapshots as image files (e.g., PNG). These visualizations help verify source detections and provide spatial context, integrating seamlessly with other outputs for final reporting and interactive review.

Classes:

- `PostageStampPlotter`: Manages the generation and export of postage stamp plots.

5.9.1 PostageStampPlotter Class

This class orchestrates the process of generating postage stamp images, including image loading, scaling, and visualization.

`__init__()` Initializes the postage stamp plotter with the required configuration.

```
def __init__(self, config):
    """Initializes the PostageStampPlotter with configuration settings."""
    self.config = config
```

`load_image()` Loads a FITS image file for processing.

```
def load_image(self, image_file):
    """Loads a FITS image file and returns the data array."""
    with fits.open(image_file) as hdul:
        image_data = hdul[0].data
    return image_data
```

`crop_region()` Extracts a postage stamp region from a larger image.

```
def crop_region(self, image_data, center_x, center_y, size):
    """Extracts a postage stamp region from the full image."""
    half_size = size // 2
    return image_data[center_y - half_size:center_y + half_size,
                      center_x - half_size:center_x + half_size]
```

`apply_scaling()` Applies logarithmic scaling to enhance visibility.

```
def apply_scaling(self, image_data):
    """Applies logarithmic scaling to enhance image visibility."""
    return np.log1p(image_data)
```

`plot_image()` Generates and saves a postage stamp plot.

```
def plot_image(self, image_data, output_file):
    """Generates and saves a postage stamp plot."""
    plt.figure(figsize=(5, 5))
    plt.imshow(image_data, cmap='gray', origin='lower')
    plt.colorbar()
    plt.savefig(output_file)
    plt.close()
```

`export_plot()` Exports the postage stamp plot as an image file.

```
def export_plot(self, image_data, output_directory, filename):
    """Saves the postage stamp plot to the specified directory."""
    output_file = output_directory / f"{filename}.png"
    self.plot_image(image_data, output_file)
    print(f"Postage stamp plot saved to {output_file}")
```

5.10 output_template.jinja

This is a template that defines how the final report is formatted in HTML. It takes the processed data and arranges it into a clear, readable webpage.

Technical Overview: `output_template.jinja` is a Jinja2 HTML template that defines the structure and layout of the final output report. It is used by `exporter.py` to dynamically render the processed data—such as light curves, source details, and visualizations—into a cohesive, user-friendly webpage. By separating the presentation layer from the processing logic, the template allows for easy customization of the report's look and feel, ensuring that the final output is both informative and visually appealing. This formatted output is subsequently served by `server.py` for interactive viewing and further user engagement.

5.10.1 Template Structure

The template consists of HTML and Jinja2 template syntax that dynamically inserts content from the processed data. It is divided into sections for metadata, table formatting, and interactive elements.

Header Section: This section contains metadata and stylesheet definitions for formatting the rendered HTML page.

```
<head>
    <title>Light Curves</title>
    <style>
        body {
            font-family: 'Roboto', sans-serif;
            display: flex;
            flex-direction: column;
            align-items: center;
            font-size: calc(10px + 2vmin);
            zoom: {{ zoom }};
        }
    </style>
</head>
```

Dynamic Content Rendering: The main content section dynamically inserts source names, observation data, and threshold values based on the processed dataset.

```
<body>
    <h1>Light Curve Output for {{ object_name }}</h1>
    <p>
        Significance Threshold:    {{ significance_threshold }}<br />
        Counts Threshold:    {{ counts_threshold }}<br />
        Search Radius: {{ search_radius }} arcmin<br />
        Sources: {{ source_count }}
    </p>
```

Table of Contents Generation: The template iterates through available sources and observations to generate a structured list.

```
<div class="table-of-contents">
    <ul class="source-list">
        {% for source_name, source_data in master_data.items() %}
        <li>
            {{ source_name }}
            <ul class="observation-list">
                {% for observation in source_data %}
                <li>
                    <a href="#{{ source_name + '/' + observation.
                        columns['Observation Id'] }}">
                        {{ observation.columns['Observation Id'] }}
                    </a>
                </li>
                {% endfor %}
            </ul>
        </li>
        {% endfor %}
    </ul>
</div>
```

Light Curve Table: This section dynamically renders observation data in an HTML table format with color-coded significance.

```
{% for source_name, source_data in master_data.items() %}
<h2>{{ source_name }}</h2>
{% if not source_data %}
<p>No data, either invalid or unsupported.</p>
{% endif %}
<table>
    {% for observation in source_data %}
    <tbody id="{{ source_name + '/' + observation.columns['Observation
        Id'] }}">
        <tr>
            {% for column in observation.columns.keys() %}
            <th>{{ column }}</th>
            {% endfor %}
        </tr>
        <tr>
            {% for value in observation.columns.values() %}
            <td>{{ value }}</td>
            {% endfor %}
        </tr>
    </tbody>
    {% endfor %}
</table>
{% endfor %}
```

Interactive Elements: This section adds interactive elements for adjusting binsize and recalculating light curves.

```
    <div class="binsize-recalculation-container">
        <span>Binsize: <input class="new-binsize-field" type="number" />
            <button class="binsize-recalculation-button" type="button">
                Recalculate</button>
        </span>
    </div>
```

JavaScript Integration: The page includes JavaScript that enables interactivity, such as handling button clicks for recalculations.

```
<script>
    document.addEventListener("click", (event) => {
        if (event.target.tagName === "BUTTON") {
            alert("This page is not running on the server.");
        }
    });
</script>
```

6 Effective Use and Future Steps

This project is designed as a starting solution for processing and visualizing astronomical data from X-ray observations, enabling researchers to efficiently extract insights from complex datasets. Future improvements will focus on optimizing processing speed by refining algorithms and leveraging parallel processing techniques, enhancing data visualization through the integration of interactive plotting tools and dynamic dashboards, and extending support for additional datasets from observatories such as XMM-Newton and NICER. These enhancements will broaden the scope of the analysis, facilitate more comprehensive comparative studies, and ensure that the platform remains a versatile and powerful tool for ongoing astronomical research.

7 Acknowledgements

I would like to thank Dr. Rosanne Di Stefano for her valuable input and guidance as my advisor, whose support in her group has been instrumental throughout this project. I am also grateful to Vinay Kashyap for his abundant technical knowledge, as well as to Victor Long and Olivia Kay for their insightful feedback. Extensive help from online sources, colleagues, and artificial intelligence has contributed significantly to the development and refinement of this project. I hope this group will continue to develop and use this program to derive great results.