GitHub Repository [Link]

LEO STM Code Documentation

Jack Smith*[1], Maureen Cohen[2], Nick Rowell[1], Roy Williams[1]

[1] Institute for Astronomy, University of Edinburgh

[2] Formerly Heriot-Watt University

Contents

1 Introduction **2**

2 Set-up **2**

2.1 `nova_client.py` 2

2.2 `settings_template.py` 2

3 ROE Khaba Specific Functions **2**

3.1 `batch_job.py` 2

3.2 `job.sbatch` 2

4 Image Reduction & Processing **2**

4.1 `image_processing.py` 2

4.1.1 `convert_to_grey` 3

4.1.2 `cloudy_or_clear` 3

4.1.3 `circular_kernel` 3

4.1.4 `detect_streak` 4

4.1.5 `process_image` 4

4.1.6 `process_images` 6

4.2 `streak_processing.py` 6

4.2.1 `Streak` Class 6

4.2.2 `get_shutter_open_close_datetimes` 7

4.2.3 `normal_line` 7

4.2.4 `process_streaks` 7

4.3 Output Files 7

4.3.1 satellite.txt 7

4.3.2 streaks_data.txt 7

5 Satellite Identification & Analysis **9**

5.1 `identification.py` 9

5.1.1 `__init__()` 9

5.1.2 `ProcessStreaksTXT()` 11

5.1.3 `GetTLEs()` 11

5.1.4 `CutCatalogByEpoch()` 12

5.1.5 `Run()` 12

5.1.6 `RunIdentification()` 13

5.1.7 `ProcessImage()` 14

5.1.8 `AnalyseMatches()` 15

5.1.9 `IsTrailCutOff()` 15

5.1.10 `DisplayFigure()` Class 15

*Email: `jackleesmith100@gmail.com`

1 Introduction

2 Set-up

`settings_template.py` is a file with default settings for all of the other programs and `nova_client.py` sets up a connection to the astrometry.net API.

2.1 `nova_client.py`

Interface to access the astrometry.net API to acquire fits files from images.

2.2 `settings_template.py`

Script to initialise global variables which will be used throughout the program, including: file paths to other scripts and input file directories, parameters for image detection, as below:

3 ROE Khaba Specific Functions

3.1 `batch_job.py`

Code that is run to process a set of raw .NEF images and returns: cut-outs of streaklets, fits of these cut-outs, a txt file of all streaklets detected, a txt file of all streaklets which can be associated to one unique satellite transit.

This file uses both the `image_processing.py` and `streak_processing.py` scripts (see Section 4).

3.2 `job.sbatch`

-

4 Image Reduction & Processing

`image_processing.py` is first run to analyse images taken on a given night and to detect streaklets in these images.

`streak_processing.py` then runs to get information about the detected streaklets and this data is saved to a file.

4.1 `image_processing.py`

Batch processes images taken by the camera, finds satellite streaks, extracts data, and writes it to a `.txt` file.

Packages Used:

Package	Classes/Functions	Comments
settings	-	`settings_template.py` file of default parameters
os	-	
rawpy	-	
cv2	-	
datetime	-	
nova_client	-	`nova_client.py` script from Section 2.1 for astrometry.net API access
numpy	-	
scipy.ndimage.gaussian_filter	-	
astropy	wcs.WCS io.fits	
fcntl	-	

4.1.1 `convert_to_grey`

Parameters:
`rgbimage` - RGB image having been read in using `rawpy.postprocess`
Returns:
`signal` -
`var` -
`grey` -

RGB to greyscale image conversion. The function sums the three RGB values and divides by 3×255 to create grey values. Output values are between 0 and 1.

This function converts an RGB image to a background-subtracted greyscale image representing the signal from celestial sources (stars, satellites), an image quantifying the variance of each pixel in the first image, and a simple greyscale image assembled as a linear combination of the RGB channels. The construction of the background-subtracted signal image is done by assuming that each RGB channel represents a different noisy measurement of the same signal level subject to a different background level. This is an approximation as the signal in each channel will vary depending on the spectrum of the source, but it seems to provide images good enough for detecting and extracting satellite streaks.

4.1.2 `cloudy_or_clear`

Parameters:
`greyimage` - One channel of an RGB image having been read in using `rawpy.postprocess`
Returns:
Boolean - `True` (clear) or `False` (cloudy)

This function sorts greyscale images of the night sky into two categories: clear or cloudy. *Inputs: Greyscale image, upper intensity bound of background, lower intensity bound for stars, Gaussian filter sigma.*

Sorting of images into cloudy (discarded) and clear (passed on for further processing). Steps:

- Use a Gaussian filter (σ given by the variable `cl_sigma`, set at 10) to create a blurred version of the image
- Subtract blurred version from original image
- Select a 500×500 subsection of the image
- Count the number of pixels in this subsection with brightnesses less than `cl_background_thresh`. This is considered the portion of the image that is "background"
- Ignoring the background, calculate the percentage of meaningful pixels brighter than `cl_lower_thresh`
- If this percentage is greater than 0, the image is considered clear

`cl_background thresh`: We ignore the background pixels to get a more meaningful value for the percentage – otherwise it may be so small it gets rounded to 0.

`cl_lower_thresh`: Chosen based on comparison of histograms showing pixel brightness distribution for cloudy and clear nights. Visual inspection shows that cloudy nights do not have pixels brighter than this threshold.

Changing thresholds: Thresholds could be fine-tuned by trial and error. The specific values come from the pixel brightness distribution curves for cloudy and clear images.

4.1.3 `circular_kernel`

Parameters:
`radius` - Radius to use to make kernel
Returns:
`kernal` - Kernal of given size

4.1.4 `detect_streak`

Parameters:
`pixels` - Array of (x, y) pixel coordinates
Returns:
`length` - length of trail
`width` - width of trail
`a, b` - start and end points of trail (x, y)

This function computes morphological properties of the source defined by the given set of pixels, to enable streak classification. Uses for outputs: ratio of the length to width of the source, and the (x, y) coordinates of the points at each end of the source along its longest axis, allowing a margin for the PSF size.

4.1.5 `process_image`

Parameters:
`datadirectory` - Directory where files are stored
`file` - Name of the image file in the above directory
`streaks_file` - Text file with data of all trails saved in
`processed_images` - List of image names which have already been processed
`output` -
Returns:
None

- Reads in image using `rawpy.imread` and convert to RGB pixels using `.postprocess`()
- Checks if image is clear or not
- If image is cloudy, skip it
- Use `conert_to_grey` to get `signal, noise, grey` variables
- Finds sources in the image and applies morphological opening to remove noise, then morph. closing to merge streaks which are fragmented
- **Connect pixels to build sources** (`cv2.connectedComponentsWithStats`)
- Cycle through sources to find those which are streaks using `detect_streak`
  - Works by finding ratio of length to width of source - trails will be long and stars will be points
  - Gets end points of trails
- Creates array of all streaks in form (x1, y1, x2, y2) of end points
- If clear image has no streaks then adds comment in processed images file
- **Create symlink to original NEF image, so we can preserve these. Eliminate symlinks from input file path**
- Process each streak in image separately assuming they are different satellites
  - Integrates astrometry.net via API key etc.
  - FYI This does include wcs file and creating wcs (w) element which I use in my current programs for visualisation
  - Saves to streak file: filename and ra, dec, x, y of both end points of each trail
  - Adds comment to processed images file about streak being clear and number of trails identified

Processing of images and extraction of data into a `.txt` file. Steps:

- Use `rawpy` to convert RAW image data into RGB

- Convert to grey using `convert_to_grey` function
- Check if cloudy or clear using `cloudy_or_clear`, discard cloudy, continue with clear
- Scale 0-1 values up to 0-255. This is necessary for compatibility with OpenCV's data type (unsigned integer 8-bit)

> Important note about OpenCV: While Python indexes images as (rows, columns) the OpenCV library indexes them as (columns, rows). This means that if you want to manipulate a specific pixel in e.g. numpy, you would access `image[125, 250]` to get the value at $y = 125$, $x = 250$. If you want to e.g. draw a red dot on this pixel using OpenCV, you must draw it at `image[250, 125]`.

- Use OpenCV's Canny edge detector to find edges.

> The Canny edge detector identifies edges based on the image gradient. The function uses the Sobel gradient calculated within an `apertureSize` or kernel. It needs a lower and an upper threshold, `definitely_not_an_edge` and `definitely_an_edge`. The Canny function classifies pixels with a gradient lower than `definitely_not_an_edge` as not edges and sets them to black. It classifies pixels with a gradient higher than `definitely_an_edge` as edges and sets them to white. If a pixel's gradient is between these two, it is classified based on whether the neighbouring pixels are edges or not. The result is a binary black-and-white image. This pre-processing step is very helpful in getting a useful result from the Hough transform.
> **Changing thresholds:** The thresholds have been set by trial and error, but could be calculated specifically for each image based on expected ranges of gradients. OpenCV's Sobel gradient function should be used when finding values such as the average gradient, standard deviation, etc.

- Use OpenCV's `HoughLinesP` function to find lines

> The first argument is the binary input image. Second and third arguments are the resolution of the r and θ parameters. The `line_votes` argument specifies the minimum number of pixels (length $\times$ width) a line must have to be detected. `minLineLength` is the minimum length of a line and `maxLineGap` is the maximum distance allowed between segments in order for them to be considered the same line. The output takes the form `[x1 y1 x2 y2]`, the endpoints of one detected line.

- `HoughLinesP` returns the endpoints of all lines it has found as two sets of (x, y) coordinates. Average these results to get a streak.

> Because the satellite streaks are relatively wide (e.g. $20 - 50$ pixels), the `HoughLinesP` function can't tell which of the end pixels are the "real" endpoints, and returns multiple overlaid lines corresponding to the same streak. We average these values to get one set of endpoints. The code can currently find one streak per image.
> **Future development:** Create code that can find two or more streaks. This might be done using the `itertools` or `more_itertools` packages to iterate over the list of endpoints and sort them into groups based on how close to each other they are.

- Draw a box around the satellite streak and save this subsection of the image as `.png` file.
- Upload `.png` file to Astrometry.net, which returns a WCS file containing the astrometric conversion data.

> An account on nova.astrometry.net and an API key are needed for this step.

- Use Astropy's WCS package and the WCS file to convert the (x, y) endpoints into right ascension and declination in degrees.
- Calculate the slope and intercept of a line fit to the streak.
- Collect data: filename, two (x, y) endpoints, two (RA, Dec) endpoints, slope, intercept, timestamp, and two endpoint times.

> Each endpoint is associated with a time. By default, `endpointa_time` is associated with ($x1$, $y1$) and `endpointb_time` is associated with ($x2$, $y2$). This assumes all trails are moving in the "forward" direction (away from the origin); this will be checked and revised if necessary in post-processing. The `endpointa_time` is taken from the timestamp of the image. Since the shutter speed is known to be 5 seconds, the `endpointb_time` is simply 5 seconds later. These values can be modified if more accurate information about the camera timing and shutter speed is obtained.

- Write all data to a `.txt` file

4.1.6 `process_images`

Parameters:
`filelist` - List of all files within `datadirectory`
`output` -
Returns:
None

- Loops through every file in the directory
  - Loads **streaks** and list of already processed images
  - If given image is not already processed:
    - Call `process_image`

Processes all images in a directory and returns a text file containing streak data of the filename of an image containing a satellite streak, together with coordinate and timestamp information used in the next step to calculate orbits.

Processing of a list of images and maintenance of a processing record. This function creates a processing record to keep track of which filenames have already been processed. It runs through a list of images from a directory, skips images that have already been processed, and passes ones that have not to `process_image`.

The processing record also keeps track of the result of `process_image` for each input image. Successful streak detections are recorded as `clear_streak`. Cloudy images are recorded as cloudy and clear images without a streak are recorded as `clear_streakless`. If an error occurs during processing, the program writes the filename and ERROR to the processing record.

4.2 `streak_processing.py`

Packages Used:

Package	Classes/Functions	Comments
settings	-	`settings_template.py` file of default parameters (Section 2.2)
numpy	-	
datetime	-	
smtplib	-	
email	message.EmailMessage mime.application.MIMEApplication mime.multipart.MIMEMultipart mime.text.MIMEText utils.COMMASPACE utils.formatdate	

Processes streaklets identified in `image_processing.py` and determines RA, Decs of trail start/end points etc and saves to file.

4.2.1 `Streak` Class

A Streak object represents a distinct streak found in an Image object. It is characterised by the image and celestial coordinates of each end of the streak, and two times that are the opening and closing times of the shutter...

4.2.2 `get_shutter_open_close_datetimes`

Parameters:
`filename` -
Returns:
`time_open` -
`time_close` -

Gets open and close times of the shutter in terms of a *Python* datetime variable.

4.2.3 `normal_line`

Parameters:
`x1` -
`y1` -
`x2` -
`y2` -
Returns:
`theta` - Orientation angle [degrees]
`d` - Distance to origin [pixels]

Calculates the normal representation of the line from two points. Can handle vertical lines.

The orientation angle is measured anticlockwise from the x axis and lies in the range [-180:180]. The distance to origin is always positive.

4.2.4 `process_streaks`

Parameters:
`output` -
`date_str` -
Returns:
None

Processes streaks

4.3 Output Files

4.3.1 satellite.txt

Saved as part of `streak_processing.py` for all identified trails of each unique satellite, e.g: `https://github.com/NickRowell/leo_stm/blob/master/output/satellite2.txt`

4.3.2 streaks_data.txt

e.g: `https://github.com/NickRowell/leo_stm/blob/master/output/streaks_data.txt`

This is a CSV file with rows of all trails with the following properties. **Table below is transposed to fit onto page:**

FileName	005_2018-12-24_170529_A_DSC_0135.NEF
Time (HHmmSS)	170529
RA	354.5474
Dec	34.30804
x	3505.5
y	4536.5
RA	354.7587
Dec	33.30098
x	3662.5
y	4831
Trail Start	17:05:29
Trail End	17:05:34
Slope	1.875796
Intercept	-2039.1

5 Satellite Identification & Analysis

5.1 `identification.py`

Code to go through images of a given night and identify all satellites where possible. Function breakdown & flowchart below. File contains two classes - `DisplayFigure()` to plot graphs of satellites whilst being identified (not necessary for code to work) and `IdentifySatellites()` which is the main class to run the code. Functions below are those belonging to `IdentifySatellites()`, those of `DisplayFigure()` are summarised afterwards (Section 5.1.10).

Throughout the class, the **`self.`** identifier is used as the object in which variables and functions of the class are stored.

To run this script, only the `identification.py` file is needed and in the directory where it is located, the code **`from identification import IdentifySatellites`** should be run to start the identification process.

Packages Used:

Package	Classes/Functions	Comments
os	listdir	
	remove	
pandas	-	
numpy	unique	
	array	
	sqrt	
	allclose	
	arange	
	where	
	vstack	
	average	
warnings	catch_warnings	
	simplefilter	Used to prevent continuous error message when loading .fits files
	filterwarnings	
skyfield	api.load	Used to load skyfield base variables
	api.wgs84	Used to create positional variables in skyfield
	api.utc	Creates time variables in skyfield
	api.EarthSatellite	Function used to convert TLEs to orbit variables
math	degrees	
	atan	
datetime	datetime	
	timedelta	
astropy	units	
	coordinates.SkyCoord	Used to create coordinate transformations (e.g. RA Dec to Pixel x y)
	coordinates.FK5	Reference frame
	io.fits	
	wcs.WCS	Allows creation and use of wcs variables
	wcs.utils	
spacetrack	SpaceTrackClient	*Python* API created by space-track.org to allow for remote access to satellite catalogues

5.1.1 `__init__()`

Parameters:
None
Returns:
None

Initialisation function which is run when the `IdentifySatellites()` class is loaded. Global parameters are set in this function including: shutter exposure time, date of images to be processed (e.g. 2022-05-11 will process photos from the night starting in the evening of the 11th May and into the morning of the 12th May), directory locations

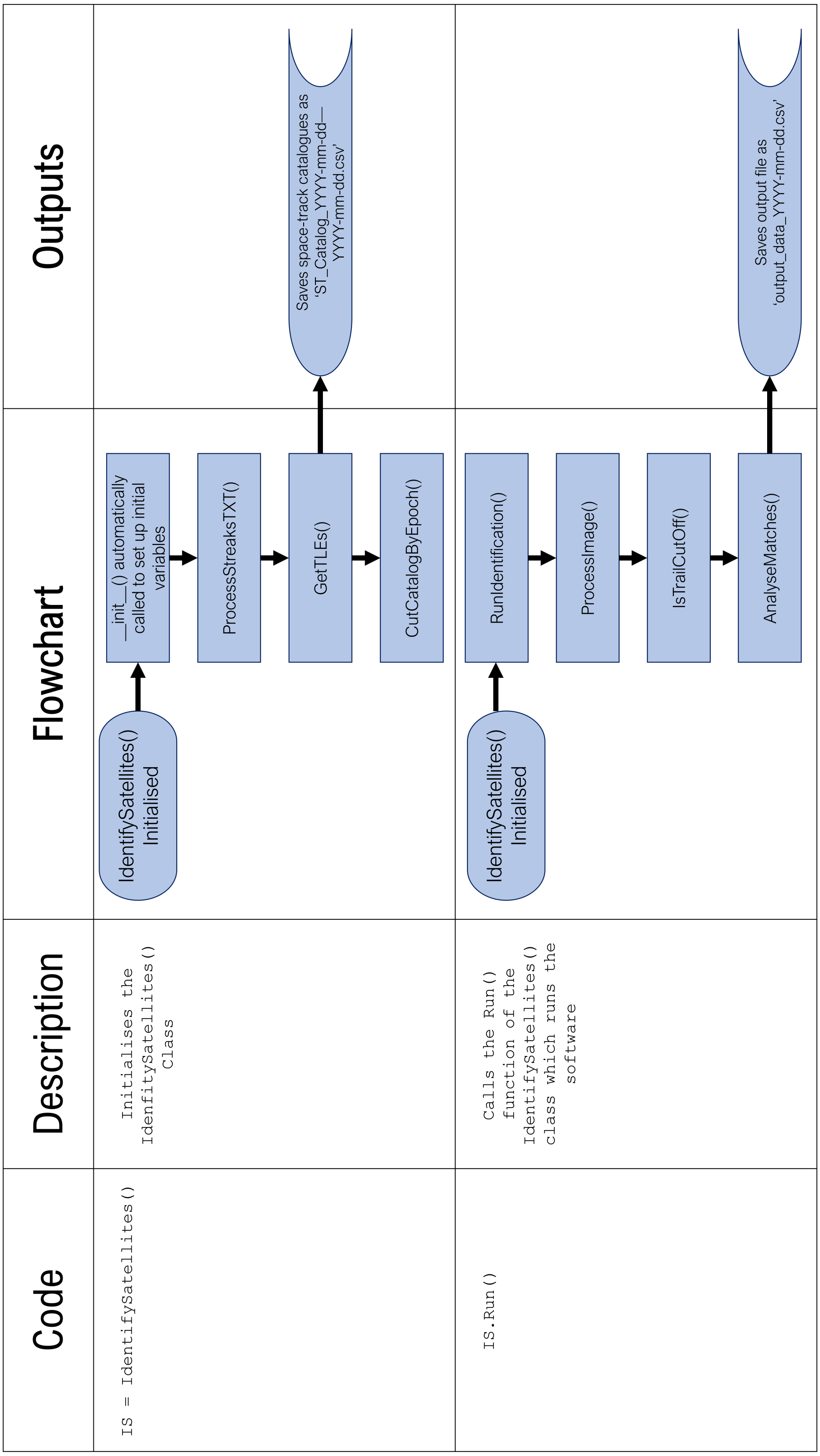

Figure 1: Flowchart of classes in `identification.py`

for fits files and .png cutouts of .NEF images from Khaba results archive. Also asks for the height and width of one of the .NEF images - this could be automated however.

`ProcessStreaksTXT()` is called to load data from `streaks_data.txt` for the given night (Section 5.1.2). Then *skyfield* is initialised as below:

```
# Initialises skyfield variables and observing position
self.ts = load.timescale()
lat, long, elevation = 55.923056, -3.187778, 146 # Royal Observatory, Edinburgh
self.bluffton = wgs84.latlon(lat,long,elevation)
```

This creates an instance of `ts` which is the timescale used by *skyfield* and a `bluffton` variable which represents the position of the observer - in this case from the ROE.

Finally, `GetTLEs()` is called to load the catalogue of satellites/objects from space-track.org (Section 5.1.3).

5.1.2 `ProcessStreaksTXT()`

Parameters:
None
Returns:
None

The `streaks_data.txt` file (Section 4.3.2) contains data in rows for each of the streaklet images which is read into a *pandas* dataframe called `trails` in this function. A short section of code is run in order to adjust the names of the files within the `trails` dataframe such that images with multiple streaklets can have their streaklets uniquely identified:

```
filenames = self.streaklets['Filename'].to_numpy()
filenames_unique, fn_counts = unique(filenames, return_counts=True)
offset, iter = 0, 0
while iter < len(fn_counts):
    val = fn_counts[iter]
    for k in range(val):
        x = iter + k + offset
        self.streaklets.at[x, 'Filename'] =
            str(self.streaklets['Filename'][x].replace('.NEF','_streak_'+str(k+1)+'.png'))
    offset += val - 1
    iter += 1
```

5.1.3 `GetTLEs()`

Parameters:
None
Returns:
None

A catalogue is produced by combining multiple API queries from space-track.org. The website provides historical TLE sets for each date, however, only TLE sets which were produced on that date are listed. As such, TLE sets for the preceding two weeks are loaded to be combined into one catalogue - two weeks is chosen since this is approximately the timescale on which TLEs become inaccurate over. Since space-track's catalogues are updated every time a new TLE is calculated, there can be multiple sets of TLEs for on unique satellite or object, with each one having its own epoch – the time at which a given TLE is most accurate. As such, for satellites with multiple TLEs over the two-week period prior to the observation, the TLE with an epoch closest to the capture time of an image is used – this is automatically calculated by the software, as described in Section 5.1.4.

Firstly in `GetTLEs()`, all of the files in the working directory are obtained. A for loop is iterated through for the number of days of past TLEs to download (14 days for the two-week span outlined above). In each iteration, the space-track catalogue for a one-day interval (e.g: 2022-07-01- -2022-07-02) is downloaded through use of the *spacetrack* package. The list of files is first checked through to determine if data for a given date interval already exists. If it does, this data is loaded and concatenated to a dataframe, otherwise a query is created using the date range and data is fetched from space-track through its API (username and password needed – free account required). Any new data loaded via the API are saved to the directory for future use and also concatenated to the aforementioned dataframe. The result is a large dataframe of all TLEs generated in the two-week period prior to (and including) the date of the images being processed.

The space-track catalogue provides numerous columns of data, two of which are utilised to restrict the objects which are used in the identification process. **Firstly, since the orbital period for LEO satellites is 128 minutes, an upper limit of 150 minutes is imposed and only objects with a period less than this are kept. (This can however be ignored such that objects in higher objects can be searched for).** Similarly, any objects which are marked as 'decayed' in the space-track catalogues (are no longer in space) are removed from the dataframe. An additional column in the dataframe is also created which is a copy of the 'EPOCH' column from space-track but in a *python* datetime format instead of a string format. The catalogue is then passed to the `CutCatalogByEpoch()` function to find the instance of each satellite with the closest epoch to midnight on the night of capture (Section 5.1.4).

The final step in this function is to pass each of the surviving satellite TLEs to the *skyfield* package which creates and saves a satellite object for each entry to the dataframe. This is the object which is later used to find the position of satellites at any given time and is unique to the *skyfield* package.

For convenience, catalogues downloaded from space-track which are for dates 3 weeks (21 days) prior to the date currently being processed by the software are deleted automatically. Thus only three weeks-worth of space-track daily catalogues will be being stored at any one time. (Each file is approximately 25 MB; about half a GB in total.)

5.1.4 `CutCatalogByEpoch()`

Parameters:
`time` - list of the date and time for which satellite TLE epochs should be closest to e.g. [2022, 7, 1, 13, 57,]
`catalog` - catalogue as loaded from space-track.org in `GetTLEs()`
Returns:
`cut_cat` - catalogue of all satellites with only one TLE which has an epoch closest to the input time

This function takes the dataframe of all satellite TLEs (remember some satellites have multiple TLEs for numerous epochs) and returns a dataframe of the same satellites but where each unique satellite has only one TLE - the one which has an epoch closest to the time provided (i.e. closest to the time at which the image was captured.)

This is achieved by first creating a new column for the time difference between the epoch of each TLE and the time provided to this function. Then, the catalog is sorted by the NORAD_CAT_ID (identification number of each satellite) and the above time difference. For each NORAD_CAT_ID, the instance of this satellite with the smallest time difference is kept and the others discarded, leaving only one instance of each satellite in the catalog, named `cut_cat`, which is returned.

5.1.5 `Run()`

Parameters:
None
Returns:
None

Call this function to run the program. Starts by creating and filling new columns in the *trails* dataframe for the start and end time of the image's capture and the gradient and y-intercept of the streaklet.

Photo#	Number from processing order of images on this night
Filename	
Satellite	Name of object (FALSE if identification failed)
NORAD_CAT_ID	ID number of object
Likelihood	Confidence in accuracy of identification
Distance	
Length	Other accuracy parameters – can be ignored, were for testing purposes
Angle	
Cut-off	TRUE or FALSE if streaklet is cut off by bounds of image
RADecPoint1 RADecPoint2	RA Dec co-ords of start and end points (1 and 2 correspond to start and end of streaklet but only where t
TLE1	Line 1 of TLE
TLE2	Line 2 of TLE

Calls `RunIdentification()` (Section 5.1.6) to perform processing and identification which returns the number of successful identifications (`num_identifications`) and `fails` which is a list of streaklets that could not be identified.

This `fails` dataframe is then looped through in order to add its contents to an output file variable (created as a global [`[`self.] variable through `RunIdentification()`, denoting these streaklets as failed identifications. This output file variable (*pandas* dataframe) is then saved to a csv with the columns:

5.1.6 `RunIdentification()`

Parameters:
`num_identifications` - number of successful identifications
Returns:
`num_identifications` - as above
`fails` - dataframe of streaklet information for those which could not be identified

An empty dataframe, `failed`, is created for streaklets which cannot be identified to be appended to. A for loop is then iterated through for each streaklet identified on this night.

For each streaklet, `ProcessImage()` (Section 5.1.7) is called to get the properties of the image and corresponding streaklet before `IsTrailCutOff()` (Section 5.1.9) is also called to determine whether the streaklet is cut off by the bounds of the image. Should at least one possible satellite be found which this streaklet could be identified as, the `AnalyseMatches()` function is called to determine which of these is indeed the satellite corresponding to the observed streaklet (see Section 5.1.8). The satellite which the streaklet is identified as is appended to the output file dataframe and the `Plot()` function of the `DisplayFigure()` class can be called here should a graph of the observed and predicted positions of this streaklet/satellite want to be saved. If no satellites are found which could be a match for a given streaklet, this row of `trails` is appended to the `failed` dataframe which is eventually returned at the end of the for loop in this function.

The number of successes (`num_identifications`) is updated in this for loop for each successful identification and then returned at the end of the function. Successful identifications allow for the direction of the satellite to be identified and such the RA Dec co-ords of the start and end points of the streaklet can be assigned and are saved in order. Unidentified streaklets have their RA Dec co-ords saved to the output csv file in an order which is not representative of their direction of travel since this cannot be determined without a successful identification.

5.1.7 `ProcessImage()`

Parameters:
`streaklet_data` - one row of the dataframe of data for each streaklet
Returns:
`w` - wcs object as loaded from the .fits file
`width` - width of the .png image cut-out
`height` - height of the .png image cut-out
`satellites_in_image` - dataframe of each of the satellites which are in the field of view of the camera at the time of capture
`one` - RA Dec co-ords of one point of the streaklet
`two` - RA Dec co-ords of the other point of the streaklet

The .fits file corresponding to a given image is loaded using the astropy fits functionality with a catch warning filter to prevent a default error message about a 'WCS transformation having more axes (2) than ...'. The header of this .fits file is accessed to obtain values for the height and width of the image.

Datetime variables for the start and end time of the image's exposure are found from the `streaklet_data` input parameter (variable which stores the `streaks_data.txt` data. An optional `shutter_offset` variable is available to provide a time offset to these times due to the unknown delay between assigned image capture and shutter opening time of the Nikon camera.

`intervals` is a list of *skyfield* time instances corresponding to each second in the 5-second exposure time (6 timestamps total). The `one` and `two` variables store the RA and Dec co-ords of both ends of the streaklet and `mid_ra_dec` representing the mid-point of the streaklet, with the differences in RA and Dec between this mid-point being subsequently calculated as `ra_diff` and `dec_diff` - i.e. the half-length of the streaklet.

The *skyfield* satellite object in each row of the catalog is used with the `bluffton` variable to get the RA and Dec of the satellite at each of the timestamps in `intervals` from the position of the observer.

```
for s in range(len(catalog)):
    # Skyfield calculations to find vectors between satellite and observer
    difference = catalog['SatelliteObject'][s] - self.bluffton
    # Gets topocentric co-ords at each time-point given
    topocentric = difference.at(intervals)
    # Finds RA and Dec values at each time-step (& convert to degrees)
    ras, decs, dists = topocentric.radec()
    ras = ras._degrees
    decs = decs._degrees
```

This section of code accounts for a reasonable percentage of run time of the program (excluding the catalogue-loading functions), due to the number of objects in the catalog being iterated through and the relatively computationally-heavy profile of the *skyfield* `difference.at` command. In this for loop, each set of RA Dec points which are obtained are passed into an if statement to determine if **any** of these points lie within one half-length of the streaklet from the mid-point in both the RA and Dec axes. If one point does, then this satellite is added to a list of satellites which are present in the camera's FOV at the given time (`satellites_in_image`). This list is converted into a *pandas* dataframe before being returned at the end of this function. It contains the columns: 'Name','x','y','NORAD_CAT_ID','TLE1','TLE2','RADecPt1','RADecPt2' where x and y are the positions of the end points of the streaklet in the .png cut-out, found using a wcs transformation between RA Dec and x, y pixel (via astropy SkyCoord).

5.1.8 `AnalyseMatches()`

Parameters:
`arr` - dataframe of possible satellite matches from `ProcessImage()`
`i` - ID number of streaklet out of all streaklets observed this night
`width` - .png cut-out width
`height` - .png cut-out height
`cutoff` - TRUE or FALSE if image cuts off streaklet
`RADecPoint1` - RA Dec co-ords of one end of streaklet
`RADecPoint2` - as above for second point
Returns:
`results` - dataframe of all satellites in the camera's FOV with their results from this function
`point1` - RA Dec of one point
`point2` - RA Dec of the other point

Creates lists for the two end points of streaklets by their x and y pixel positions of the possible satellite matches from both observations (`sat_x, sat_y`) and expected positions (`test_x, test_y`). The matching process uses all of the satellites which were found to lie within the camera's FOV at a given time to determine which one is most likely the satellite which was observed. This is done through comparison of three qualities of the expected streaklets obtained from *skyfield*: expected streaklet length, expected angle of streaklet, and distance between the expected and observed streaklet. In each case, a normal distribution is created from the observed streaklet and the values of expected streaklets compared to these distributions for each possible satellite in the FOV. A combination of these three distributions is used to provide a 'likelihood' value for each satellite, the highest indicating the satellite which is expected to have caused the observed streaklet.

The direction of motion of the satellite is easily obtained once a satellite has been identified and a dataframe of the results for each satellite in the FOV is returned from this function, sorted by likelihood.

5.1.9 `IsTrailCutOff()`

Parameters:
`i` - ID number of streaklet out of all streaklets observed this night
Returns:
`TRUE` / `FALSE`

Uses the height and width of the original .NEF files to determine if the position of either end of a streaklet is outside of the bounds of the .NEF image. Returns `TRUE` or `FALSE` as appropriate.

5.1.10 `DisplayFigure()` Class

There is an option to save a figure to a file for each streaklet which is processed. This is (optionally) called in the `RunIdentification()` function and roughly increases the time to identify each streaklet by 50%. This class uses the *matplotlib* package which needs to be imported into `identification.py` should this class want to be used. (This code is present but commented out.)

An example output from this class is shown in Figure 2 below.

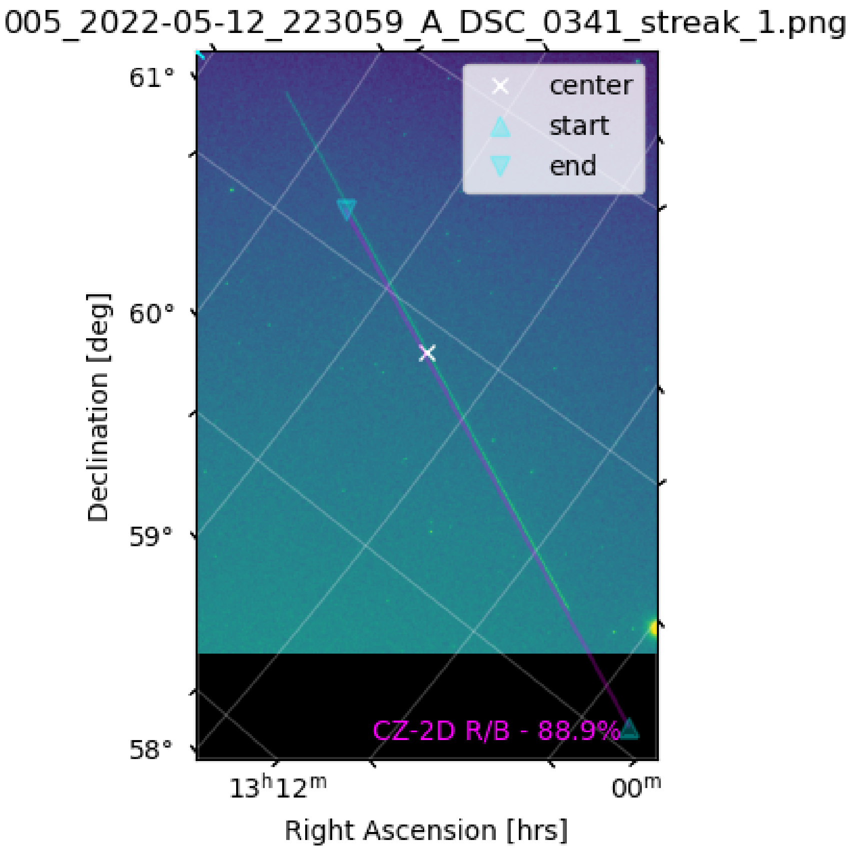

Figure 2: Example output from the `DisplayFigure()` class.