Colibri Diagnostic Scripts

Rachel A. Brown

June 2022

Contents

This document describes the diagnostic scripts used to examine Colibri data. The following scripts can all be found in RachelPipeline on the projects GitHub, or in the rbrown/Documents/Colibri/Scripts folder on Pomegranate.

A note on file structure:

I use the same general file structure on both Pomegranate and the Colibri computers. All of the scripts in this document, and the Colibri pipelines, are designed to work within this file structure.

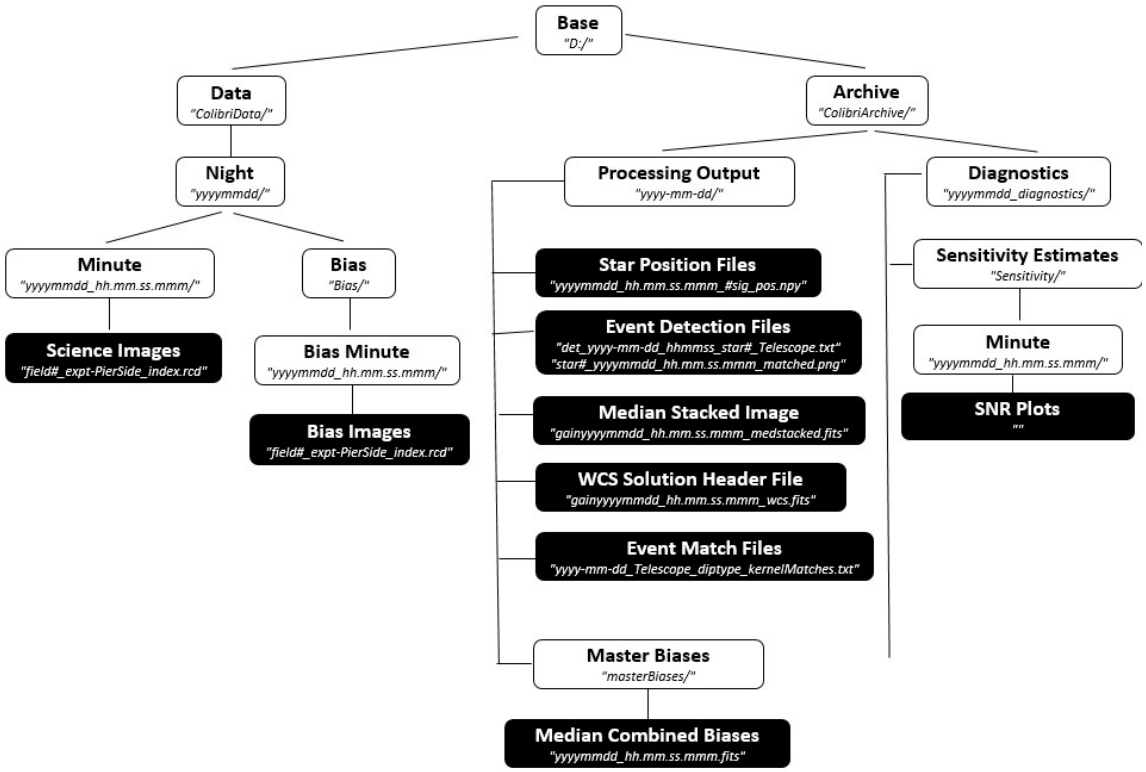

Figure 1: Tree showing the expected data file structure on each of the three Colibri telescope computers, and on Pomegranate. All scripts run on Colibri data work within this structure.

Colibri project files are stored in the *"Base"* directory. On the Elginfield computers this is the *"D:"* drive. On Pomegranate, this is */rbrown/Documents/Colibri/**telescope***, where ***telescope*** is either *"Red"*, *"Green"*, or *"Blue"*. Data for the project is divided in two categories: *data* (*"Base/ColibriData/"*) and *archive* (*"Base/ColibriArchive/"*). All the scripts below have a *base_path* variable that can be changed to the *base* directory for any file system. As long

as the data and archive files are arranged with the same structure described below, the scripts should run on other systems by changing only this path.

The *data* folder contains the science and bias images and is further subdivided by observation *night* (ie *"20210804/"*) and *minute* (ie *"20210804_05.03.22.121/"*). Bias images are stored in a *bias* folder in the *night* directory, and are also subdivided by *bias minute*.

Archive files include any of the pipeline outputs or diagnostic script outputs. Pipeline outputs (star position files, detection files, median combined biases) are stored in an *output* folder, labelled by the date of processing (ie *"2022-06-16"*). Diagnostic outputs (sensitivity estimates, timestamp checks, bias subtracted images etc) are stored in a *diagnostics* folder, labelled by the date of observation (ie *"20210804_diagnostics*). This is further subdivided by the diagnostics performed (ie sensitivity estimates, timing tests, median combined biases, bias subtracted single images). Some of these are further subdivided by *minute* to avoid mixing up diagnostics taken at multiple times throughout a single *night*.

include example plots, table outputs?

1 astrometry.py

This is used to analyze plate solutions from astrometry.net. This script does not query astrometry.net directly like **astrometry_funcs.py** (Section **??**). You will need to upload the image to the astrometry.net website and download the output files manually from there.

1.1 Need to Run

The following should be saved in an archive folder labelled with a timestamp that matches the timestamp of the original data folder (usually in the night's *Sensitivity* directory).

- Set of output .fits files from astrometry.net. These are the files that can be downloaded after using their web service. These include:
  - **WCS header file**. This contains the WCS solution calculated by astrometry.net (called *wcs.fits*). This should be saved as *filename_**index**_wcs_**order**.fits*, where ***index*** is the index number of the image (ie *"0000001"*,*"0002400"*) and ***order*** is the tweak order of the solution.
  - **Annotated image file**. This is original image with the WCS solution embedded (called *new-image.fits*). This should be saved as *filename_**index**_new-image_**order**.fits*.
  - **Reference stars table**. These are the stars astrometry.net uses from its catalogs to calculate the solution (called *rdls.fits*). This should be saved as *filename_**index**_rdls_**order**.fits*.
  - **Detected stars table**. These are the stars astrometry.net detected in the image (called *axy.fits*). This should be saved as *filename_**index**_axy_**order**.fits*.
  - **Correspondence table**. This is the corresponence between the stars detected by astrometry.net and its reference stars (called *corr.fits*). This should be saved as *filename_**index**_corr_**order**.fits*.
- Table of stars detected by **sep** in the main Colibri pipeline (or **Sensitivity.py**, Section 9). This should be a .npy file with the first 2 columns containing X and Y coordinates.

1.2 Algorithm

1. Set up the observation and solution info. This includes the observation date, time, the index of the image that was uploaded to astrometry.net (for example, if you uploaded *"field1_25ms-E_0001000.fits"* the index is *"0001000"*). The tweak order of the solution that was obtained, the telescope name, and the name of the field observed are also defined.
2. Filepaths are set up. The tolerance in pixels for matching tables is defined as well.
3. Star coordinates from **sep** run on the Colibri stars are transformed into [RA, dec] using the solution from **astrometry.net**. This is done using the function **getRADecfromFile** in **getRADec.py** (Section **??**). This saves the transformed coordinates in a .txt file that is reloaded into the script as a dataframe.
4. The output files from astrometry.net are read into the script using **getResults** (Section **??**). The correspondence table (see Section 1.1) is made into a dataframe.

5. The Colibri stars and **astrometry.net** stars are matched using **match_XY** (Section **??**).
6. If **sensitivity.py** has been run previously, load in the results table to get Gaia data for the detected and matched stars. Match this table with the original table using **matchColibriAstnet** (Section **??**). If this hasn't been run, this section can be commented out, along with the line to plot Gaia vs Astrometry.net coordinates.
7. A set of plots comparing solution coordinates are generated.

1.3 Required Modules

- pathlib
- astropy
- numpy
- matplotlib
- datetime
- pandas
- getRADec.py

2 astrometrynet_funcs.py

This makes use of the **astrometry.net** API, allowing a query to be made directly from a script instead of by uploading and image to the webpage. It contains a single function.

2.1 getSolution

- Sends request to **astrometry.net** API to get a WCS solution for an image.

Input:

1. Filepath to the image file that is being solved [pathlib Path object, or string]
2. Filepath to the .fits header file that the solution will be saved to [pathlib Path object or string]
3. Tweak order for the solution [int]

Returns:

1. WCS solution header. Also saves this to a .fits file.

Algorithm

1. The API is set up with the key for the user's account.
2. A query is made using the desired parameters (tweak order, centering etc) and the WCS solution header is returned.
3. The header is saved to a .fits file if one does not already exist.

2.2 Required Modules

- AstrometryNet from astroquery.astrometry_net

3 bias_subtract.py

Subtract a median-combined bias image from a single science image, and save the new image as a .fits file. Records statistics (mean, median, RMS) for the new image in a .txt file.

3.1 Algorithm

1. Set up the parameters to run the script. It can run on .fits or .rcd files. The image index is the identifier for the image that is having the bias subtracted (ie for *'field1_25ms-E_0000003.rcd'* the index is *'0000003'*). Images can be either high or low gain.

2. Set up filepaths to find the images (science and bias), as well as the location to save the outputs to. If any of the output directories do not already exist, create them.

3. Make or load the median combined bias image for subtraction. First, the list of master bias images is loaded in. If there are no master bias images already existing, the a median stack is created using **makeBiasSet**. If the biases already exist, we need the timestamps for them to determine which is the best one to subtract. This is done using the **getDateTime**.

   Once these steps are done, *biasList* is a 2D array containing the filepaths and datetimes of each master bias image. This is passed to **chooseBias**, which selects the bias image that is closest in time to the science image.

4. The image is read in and the median combined bias subtracted from it. If the image is in .rcd format, this is done through **importFramesRCD**, which also subtracts off the bias it is passed. The final subtracted image is saved to a .fits file.

   If the image is in .fits format, it is read using the **astropy fits** module. The bias is then subtracted, and the modified image saved to a new .fits file.

5. The image statistics are calculated and saved. Sigma clipping is performed to remove the effect of stars on the image statistics. Statistics without sigma clipping are calculated as well. These are all printed to the console as output. The sigma clipped stats are saved in a .txt file containing statistics info for all the bias subtracted images in a night. If this file doesn't already exist, it is created with the first line as a header. Statistics are then appended to this file.

4 getRADec.py

- Contains a set of functions to transform coordinates from (X,Y) to (RA,dec) using a World Coordinate System (WCS) solution header.
- Used by several other scripts (including the secondary pipeline) to transform coordinates.

4.1 Required Modules

- astropy
- numpy
- pathlib
- datetime

5 image_stats_bias.py

Gets image statistics (mean, median, RMS) for each bias image taken in a night, as well as the sensor temperatures. Saves these in a text file.

5.1 Need to Run

- Bias images saved in *minute* directories, within the *bias* directory in a *night* directory. These should be in .rcd file format.

5.2 Algorithm

1. The observation information (telescope name, observation date, and gain level) are defined.
2. Paths to the image directories and to the saved text file are defined. The text file is created with a column header as the first line.
3. Each *minute* in the bias directory is looped through.
   (a) A list of images within the current *minute* directory is obtained.
      i. Each image within the list is looped through. The image data, including pixels values, header time, and sensor temperatures, is read using **importFramesRCD** (Section **??**).
      ii. The image array is flattened from a 2D array to a 1D array. This is to speed up the statistics calculations.
      iii. The image statistics are calculated. This is done using the **numpy** mean and median functions, and the **scipy.stats** mode functions. The root-mean-square is also calculated as

$$RMS = \sqrt{\bar{x^2}} \tag{1}$$

      iv. These quantities are all appended as a line in the output text file.

5.3 Functions

5.3.1 importFramesRCD

- Reads in multiple images from .rcd files in the directory.
- Adapted from **importFramesFITS** by Mike Mazur (See Main Colibri Pipeline documentation).
- Slightly modified from other versions, also returns sensor temperatures.

Input:

1. List of image filepaths in the data directory [list of path objects]
2. Starting frame number [int]
3. Number of frames to read in [int]
4. Bias image [2D array of counts]. Must be the same size as the images in the directory.
5. Gain keyword [string - either *"high"* or *"low"*]

Returns:

1. List of image data arrays [list of floats]
2. Array of header times for these images [1d array of strings]
3. Tuple of three sensor temperatures: sensor temperature, base temperature, FPGA temperature.

Algorithm:

1. Create empty lists to hold the image arrays and header times. Also set the width and height of the images (in px) manually.
2. Make a list of filenames to read in. This is done by list comprehension; taking the input list values that are between the given starting index and the starting index plus the given number of frames to read.
3. Loop through each image filepath in the new list.
   (a) Open the image using the **readRCD** (Section **??**), which returns both pixel data and header. Get the timestamp of the image, as well as the three sensor temperatures.

(b) Get an image from the pixel data using **nb_read_data** (Section **??**. The data contains both high and low gain pixel counts, so it then needs to be split using **split_images** (Section **??**).

(c) Check if there is a time error and correct it. In some cases during testing, the header timestamp would read a time of "29:00h", which of course is an error. This section of the code was written to identify the issue and patch it so we can continue processing.
The header time string is separated out to get the hour and minute of the image's timestamp. The minute the parent directory was created is taken from the directory filename. If the hour is greater than 23, we have an issue.
The correct hour is taken from the parent directory name. If the image timestamp minute is less than the directory name minute, the hour must have rolled over during the minute of observations. In this case, 1 will needed to be added to the correct hour. Note that during the process of developing the pipeline, only Red computer was using UTC. At the time of writing this documentation, both Red and Green use UTC. Since all image timestamps are in UTC, any telescope computer that is using local time will need to add an additional 4 (5 if daylight savings) hours to the corrected time. These lines are commented out in the script.
The incorrect hour is replaced by the corrected one in the timestamp string.

(d) The image is closed. The image array is appended to the list of images, and the timestamp string appended to the list of header times.

4. The list of images is made into a multidimensional array. This is for ease of operations later on in the pipeline.

5. The data is reshaped if necessary, and count values are made into floats.

5.3.2 readxbytes

- Written by Mike Mazur.
- Read in the specified number of bytes (for reading .rcd images).
- Always used after the file has been opened, and the **seek** function has been used to read until the starting byte.

Input:

1. number of bytes [int]

Returns:

1. data in the specified number of bytes[array]

5.3.3 nb_read_data

5.3.4 getSizeRCD

- Get the number of .rcd images in a data directory (for minute-long data sets should be around 2400). Get the dimensions of .rcd images.
- written by Mike Mazur.

Input:

1. List of image filepaths in the data directory [list of path objects]

Returns:

1. width of .rcd image [px]
2. height of .rcd image [px]
3. number of .rcd images in directory [int]

Algorithm:

1. Get the number of images in the directory by finding the length of the list of image filepaths.
2. Open the first image header in the directory to get the X, Y axis dimensions.

5.4 Required Modules

- numpy
- numba
- scipy.stats
- binascii
- datetime
- pathlib

6 lightcurve_looker.py

- Functions for making plots of either entire light curves, or single events.

6.1 Functions

6.1.1 plot_wholecurves

- Plots a light curve for each star for an entire minute-long dataset.

Input:

1. Directory containing .txt files with the light curve for each star.

Returns:

1. .png files showing the light curve for each star in the minute.

6.1.2 plot_event

- make .png file plotting the portion of a star's light curve which contains an event.

Input:

1. directory containing detection .txt files.
2. dataframe containing the star's data (flux and time values)
3. frame number of the event
4. ID number of the star
5. Coordinates of the star [X px, Y px]

Returns:

1. .png file showing a plot of the event and surrounding light curve.

7 lightcurve_maker.py

Modification of colibri_main.py that identifies stars in a set of images, and saves the light curves as .txt files. It is written as a series of functions, and is designed to be called by another script instead of being standalone. The main function for this script is called **getLightCurves**.

7.1 Need to run

- Directory containing .rcd or .fits images ordered by time and labelled sequentially (*folder*). Expected file structure is *'**yyyymmdd**/**yyyymmdd_hh.mm.ss.mmm**/'*.
- A set of bias images for the night's observations. Expected file structure is *'**yyyymmdd**/Bias/'*.

7.2 Functions

7.2.1 getLightCurves

- Detect stars in the image set and save their light curves as .txt files

Input:

1. Path to folder containing dataset [path object]
2. Path to folder to save .txt files in [path object]
3. Aperture radius for photometry [px]
4. Gain level for .rcd images ['*low*' or '*high*']
5. Telescope name [string]
6. Detection threshold level for star finding [float]

Returns:

1. Directory of .txt files for each star

8 read_npy.py

Reads the star position .npy file and makes a plot of the star positions. Also contains a function to translate the star positions into a .txt file that can be read by ds9 and plotted over a .fits image as a region.

9 sensitivity.py

This script is used to make plots of Gaia G-band magnitude vs Colibri SNR for detected objects.

9.1 Need to Run

- Directory containing .rcd or .fits images ordered by time and labelled sequentially. Expected file structure is *'**base_path**/ColibriData/ **Telescope**Data/**yyyymmdd**/**yyyymmdd_hh.mm.ss.mmm**/'*, although this can be adjusted by changing the *base_path* and *data_path* variables. Output files from the script are saved in a directory that the script creates called *'**base_path**/ColibriArchive/Sensitivity/**Telescope**/**yyyy-mm-dd**/'*.
- Bias images in .fits or .rcd format (preferably multiple because *'lightcurve_maker.py'* takes a median of 10. These are expected to be in *'**base_path**/ColibriData/ **Telescope**Data/**yyyymmdd**/Bias/'*
- *as of Feb 7 2022 you also need an output file from astrometry.net. It is the output file labelled as emph'new-image' when you use the web version of astrometry.net. The filename for this should contain the original image index number that astrometry was performed on, *'new-image'*, and the order of the solution (ex *'3rd'*, *'2nd'*)
- The other scripts listed in Section 11.4 in the same folder as sensitivity.py.

9.2 Algorithm

1. The first part of the script (marked '*Set up*') sets up the variables and filepaths that will be used later on. Colibri data is generally first organized by telescope, then by observation night, then by observation minute.

   The first block of code the user inputs the observation date, observation minute, index of the image used in astrometry.net, polynomial order of the astrometry.net solution, radius of photometry aperture [px], gain level, telescope name, field name, and sep.extract detection level. These options are all available so different combinations can be easily changed tested without having to go into the many different functions and change them manually.

   The next few blocks of code set up the directory structure that the script will use to find data files and save output files. It will also make output directories if needed.

2. Next object detection and photometry is done on the data using functions in *lightcurve_maker.py* (Section **??**). This will also save a median combined master bias image (in a folder labelled *'**Gain**_masterBiases/*). A median-combined, bias-subtracted image used for object detection is also saved, labelled ***Gain**_medstacked.fits*. A list of detected object coordinates from sep.extract is saved in a numpy array file labelled ***fieldname_date_time_gain_threshold**sig*. For each detected object a .txt file containing a table of fluxes from each image is saved in a folder labelled ***Gain_threshold**sig_lightcurves*. Plots of these lightcurves are made using *lightcurve_looker.py* (Section 6) and saved in the same folder as the lightcurve.txt files.

   This step takes the longest (can be several minutes depending on the number of images and detections). If I've already performed photometry with the desired parameters I'll comment the above two calls before running the script to save time.

3. The image is uploaded to astrometry.net to obtain a WCS solution. Previously this was done manually by uploading a subtracted image to the astrometry.net web service, then downloading the new image file. This can now be done automatically using a function in **astrometry_funcs.py**.

   The filepath to the median combined image saved by the previous step is obtained. The path to the file which will contain the WCS solution is also defined.

   A check is performed to see if this file already exists (it can take a long time to get a new solution every time, so if one has already been calculated for this minute it's faster to use that one). If the file already exists, it is opened and the transformation is taken from the header.

   If the file doesn't already exist, the function **getSolution** from **astrometry_funcs.py** is called. This calculates the new transformation, saves it in a file labelled ***date_time**_wcs.fits*, and returns the solution to the main script.

4. A transformation of object coordinates from (X, Y) [px] to (RA, Dec) [deg] is performed. This is done using a plate solution from astrometry.net and a list of star coordinates from the above .npy file.

   The first block of code sets up the filenames for this step.

   The next block performs the coordinate transformation on the object coordinates using functions in *getRADec.py* (Section 4), and makes the output into a pandas dataframe for ease of use (*coords_df*). A table listing object coordinates in (X, Y) and (RA, dec) is saved to a .txt file labelled *XY_RD_**gain_threshold**sig_**date_time**.txt*. In this step, a file is saved with star coordinates in a format that can be read by DS9 using a function in *read_npy.py* (Section 8). This file is labelled ***fieldname_date_time_gain_threshold**sig_pos_ds9.npy*. To use this file, open a .fits image in DS9, go to *Regions*, *Open Regions* and select the file. Objects are marked using circles over the image.

5. A query is made to the Gaia catalog to get all star data (coordinates and magntiudes) for stars within a search radius from the field centre that are also within a magnitude limit. This is done using a function in *VizieRQuery.py* (Section 12). Stars are returned in a dataframe called *gaia* that has RA, dec, G-band magnitude, and (B-R) colour.

6. The airmass, altitude, and azimth of the observed field is calculated. The first star detection .txt file is read in to get the header time of that event, which will be used to determine the telescope's pointing. This header time is passed along with the field centre coordinates to the function **getAirmass** (Section 9.3.1) to get the airmass, altitude, and azimuth of the observed field.

7. For each detected object the median flux, flux standard deviation, and SNR (median/stddev) is calculated and returned in a dataframe containing the object coordinates and these values. This is done using a function in *snplots.py* (Section **??**). A table containing this data is also saved in a file labelled *stars_snr.txt* found in the lightcurve folder.

8. There are now three tables (pandas dataframes) that need to be matched: 1. object coordinates, 2. Gaia star data, 3. object flux data. This is done in two steps. First, a dataframe containing object (X, Y, RA, dec, Gmag) called *rd_mag* is created using *match_RADec* (Section 9.3.2). This is then matched with object flux data using *match_XY* (Section 9.3.3) to create a called *final* with (X, Y, RA, dec, GMag, median flux, flux standard deviation, SNR). *note that this whole algorithm could be simplified by one step by getting the RA dec transformation directly from the object data table. This requires some changes to the structure of the code.

   If the image has any region with artefacts (for example the stripes on the edge of the image) detections in these regions can be filtered out of the final dataframe.

   This final dataframe is saved to a .txt file labelled *starTable_**minuteDirectory_gain_solutionOrder_Telescope_detection*

9. A plot showing instrumental magnitude (-2.5(log(median)) vs Gaia magnitude is displayed and saved to *magvmed_**gain_minuteDirectory_detectionThreshold**.png*.

   A plot showing Signal/Noise (median/stddev) vs Gaia magnitude is displayed and saved to *magvSNR_**gain_minuteDirecto*r

10. Prints out how many objects were detected, matched with Gaia stars, and have a SNR greater than or equal to 10.

11. Makes a scatter plot of right ascension and declination differences between Colibri data and Gaia data (Section 9.3.4).

9.3 Functions

9.3.1 getAirmass

- calculate airmass of the given field at the given time.

input:

1. header time in *isot* format [string]
2. RA of field [degrees]
3. dec of field [degrees]

returns:

1. airmass
2. altitude [degrees]
3. azimuth [degrees]

Algorithm:

1. The latitude and longitude of the site are defined.
2. The Local Sideral Time (LST) of the observations are calculated using the method **sidereal_time** from the *Time* package.
3. The Hour Angle of observations is calculated by subtracting the field RA from the LST.
4. The altitude of the field is calculated using the following equation:

$$alt = arcsin(sin(\delta) \times sin(lat) + cos(\delta) \times cos(lat) \times cos(HA)) \tag{2}$$

   Note that the **numpy** trig functions take angles in radians, not degrees so conversions are necessary.

5. The azimuth of the field is calculated. First the quantity A is calculated with the following formula,

$$A = acos\left(\frac{sin(\delta) - sin(alt)sin(lat)}{cos(alt)cos(lat)}\right) \tag{3}$$

   If $sin(HA)$ is negative, then the azimuth is equal to A. Otherwise, the azimuth is $360.0° - A$.

6. The Zenith Angle is calculated by subtracting the altitude from 90°.
7. The airmass is calculated using the following formula:

$$airmass = \frac{1}{cos(ZA)} \tag{4}$$

9.3.2 match_RADec

- matches list of Colibri detections with Gaia catalog by RA/dec to get magnitudes
- Adapted from Mike Mazur's matching script
- If multiple matches are found within the search box, the match with the brightest magnitude is selected. There are two lines that can be uncommented to select based on closest distance instead.

input:

1. pandas Dataframe of detected star data x, y, RA, dec
2. dataframe of Gaia data ra [deg], dec [deg], magnitude, colour
3. search box half side length [deg] (ex. SR = 4, search rectangle is 8 x 8)

returns:

1. original star data frame with gaia magnitude columns added where a match was found X, Y, ra, dec, Gmag

9.3.3 match_XY

- matches list of Colibri detections with Gaia magnitudes to list of Colibri detections with flux info based on X, Y coords
- Adapted from Mike Mazur's matching script
- If multiple matches are found within the search box, the match with the highest SNR is selected (in theory this shouldn't be needed, because the X and Y coordinates from the two dataframes should match exactly as they come from the same source).

input:

1. pandas dataframe of detection data with Gaia magnitudes x, y, RA, dec, G magnitude
2. dataframe of star data with SNR x, y, med, std, snr
3. search box half side length [px] (ex. SR = 4, search rectangle is 8 x 8)

returns:

1. original detection data frame with flux data columns added where a match was found X, Y, ra, dec, GMAG, med, std, SNR

9.3.4 RAdec_diffPlot

- makes scatter plot of difference between Colibri star coordinates and Gaia star coordinates
- Originally from astrometry.py (Section **??**)

input:

1. pandas dataframe of star data including an RA difference column ('*ra_diff*') and dec difference column ('*dec_diff*'). Columns should be Colibri - Gaia to match the plot labels

returns:

1. Saves scatter plot of dec difference vs ra difference, with the 2 sigma clipped mean and standard deviation for both.

9.4 Required Modules

- numpy
- matplotlib.pyplot
- pandas
- datetime
- astropy, astropy.stats
- pathlib
- getRAdec.py (Section 4)
- snplots.py (Section **??**)
- lightcurve_maker.py (Section **??**)
- lightcurve_looker.py (Section 6)
- read_npy.py (Section 8)
- VizieR_query.py (Section 12)

10 snplots.py

Contains a function to calculate the median, standard deviation, and temporal SNR (median/stddev) for a star's light curve.

10.1 snr_single

- Calculates the median, standard deviation, and SNR for a star using a .txt file containing its lightcurve.

Input:

1. Path to directory containing lightcurve text files. The input file should have a column labelled as 'filename', 'time', and 'flux'. [pathlib Path object]

Returns:

1. An array containing each star's x coordinate, y coordinate, median flux, stddev of flux, and SNR (flux/stddev).

Algorithm:

1. A set of empty lists are created to hold all the star's data. Lists of the text files are created from the directory.
2. Each file is looped through. The X and Y coordinates of the star are taken from lines in the file header. The flux and times are read in as a dataframe. The median, standard deviation, and temporal SNR are calculated and appended to the lists.
3. A .txt file containing the data for each star is saved to the same directory the individual star files are found in.
4. Star data is packaged in an array and returned.

11 timingplots.py

11.1 Need to Run

- Directory of observations for a single night (*night directory*). This should contain multiple subdirectories (*minute directories*), which in turn contain images for that minute. At 40 fps, should be around 2400 images in the directory. Images should be in .rcd format.

11.2 Algorithm

1. Set up input and output directories. The input directory (*night directory*) is given as an argument in the command line when running the script. Note that the output directory (where script results will be saved) is created relative to this. Printouts from the script will be saved in a .txt file in the output directory, along with any plots produced.

2. Each *minute directory* is looped through, checked for anomalies, and a plot of timestamps is produced.

   (a) A list of all images in the *minute directory* is created.

   (b) If this list is empty, a line is written to the output file saying that the directory is empty. It moves on to the next *minute directory*.

   (c) Each image is looped through.

      i. The image data is read using **readxbytes** (Section 11.3.1). The timestamp of the image is taken from its header. Note that the starting byte may need to be changed in future if the format of the image header is changed by FLI.

      ii. The hour and minute of observation are taken from the timestamp. This is done using a **try** and **except**, because if the file is corrupted in some way it will cause an error. In this case, we'd like to note the corrupted file and carry on with the next image.

      iii. A check is performed to see if the timestamp hour is greater than 23:59. This was happening sometimes at one point in testing (hour was showing as '29:00' which of course is nonsense). In this case, the correct hour is taken from the *minute directory* name. If the hour has rolled over during the minute, 1 will need to be added to the hour. Also, if the observing computer is in local time, then the conversion from UTC (the cameras are all in UTC) to local time needs to be made.
      The bad hour is replaced by the corrected hour once this has been done.

      iv. The frame number and timestamp of the image are added to the lists.

   (d) All timestamps are converted to Unix time so they can be compared. The list of times is then converted to a list of times elapsed since the beginning of the minute of observations.

   (e) A check is performed to test if all the image time stamps are in order. This is done because during testing we had a few incidents of "time travel", where an image would have a later timestamp than the subsequent image. We believe this issue was resolved by placing the GPS unit outside the dome. If time travel is discovered, a plot showing the event and the surrounding 100 frames is saved.

   (f) A plot showing the time elapsed since the beginning of the minute as a function of frame number is saved.

11.3 Functions

11.3.1 readxbytes

- Written by Mike Mazur.
- Read in the specified number of bytes (for reading .rcd images).
- Always used after the file has been opened, and the **seek** function has been used to read until the starting byte.

Input:

1. number of bytes [int]

Returns:

1. data in the specified number of bytes[array]

11.4 Required Modules

- sys
- matplotlib.pyplot
- astropy.time
- pathlib

12 VizieRquery.py

- Written by Jillian Ryan, modified by Rachel Brown to make the code into a function that can be called by other scripts.