Colibri Main Pipeline

Rachel A. Brown

May 2022

Contents

This document describes the main data reduction pipeline for the Colibri telescope array. The main pipeline was originally written by Emily Pass in 2018. It has since been modified by Tristan Mills (adding .fits capability) and myself. It also includes functions written by Mike Mazur to add .rcd file handling (2021). A basic overview of the pipeline flow is shown in the below flowchart:

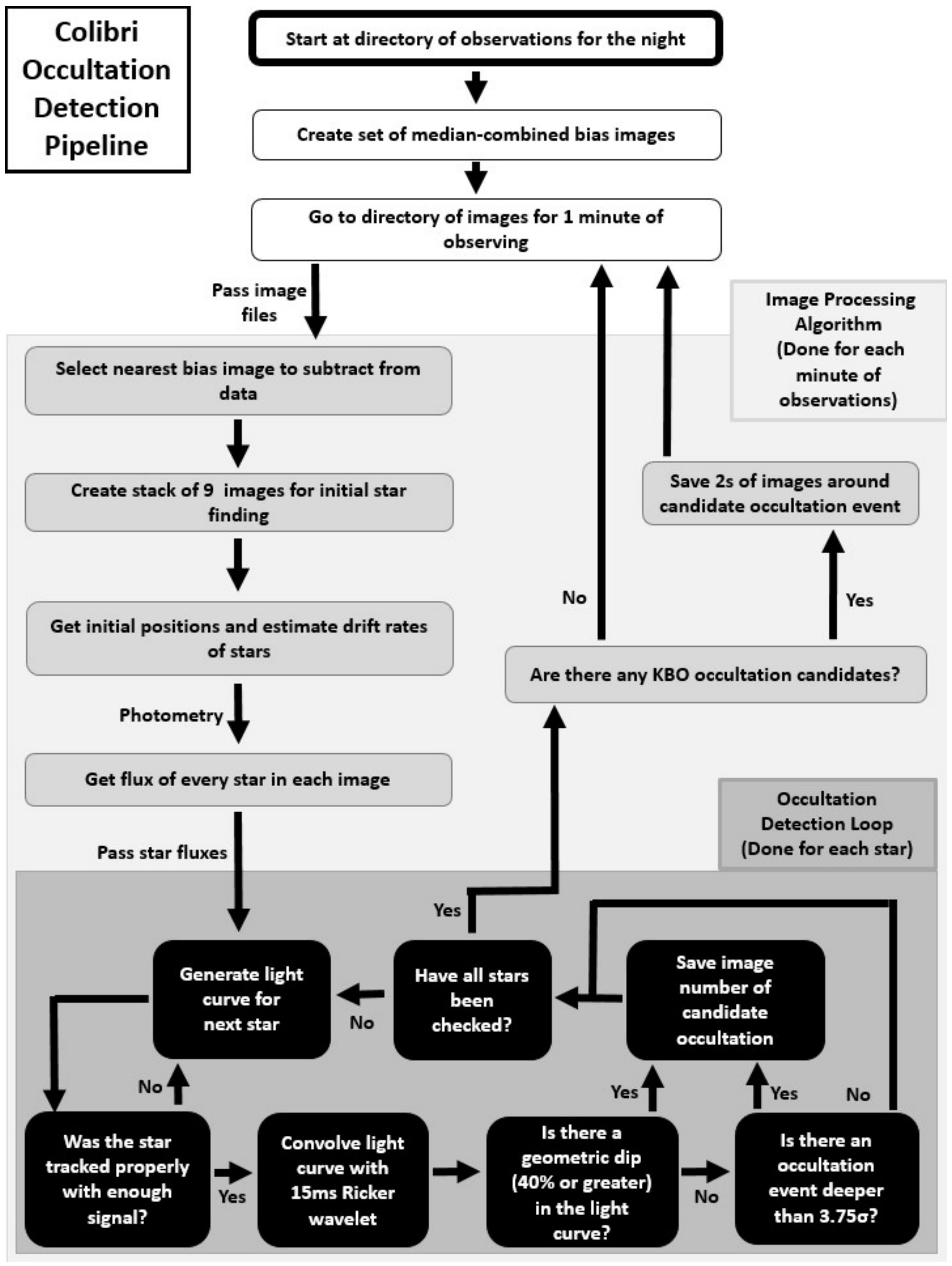
Colibri Occultation Detection Pipeline
Start at directory of observations for the night
Create set of median-combined bias images
Go to directory of images for 1 minute of observing
Pass image files
Image Processing Algorithm (Done for each minute of observations)
Select nearest bias image to subtract from data
Create stack of 9 images for initial star finding
Get initial positions and estimate drift rates of stars
Photometry
Get flux of every star in each image
Pass star fluxes
Save 2s of images around candidate occultation event
No
Yes
Are there any KBO occultation candidates?
Occultation Detection Loop (Done for each star)
Yes
Generate light curve for next star
No
Have all stars been checked?
Save image number of candidate occultation
No
Yes
Yes
No
Was the star tracked properly with enough signal?
Yes
Convolve light curve with 15ms Ricker wavelet
Is there a geometric dip (40% or greater) in the light curve?
No
Is there an occultation event deeper than 3.75σ?

Figure 1: Flowchart showing the basic steps of the primary Colibri occultation detection pipeline.

1 Need to Run

1.1 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.

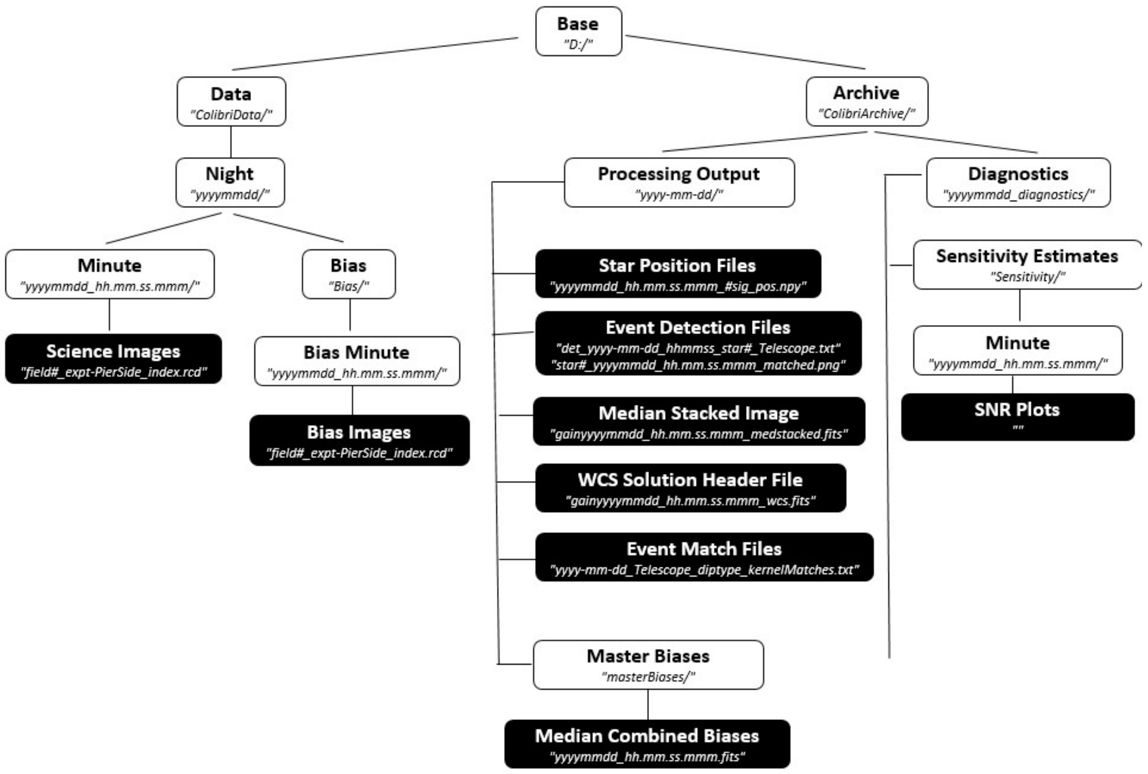

Figure 2: Tree showing the expected data file structure on each of the three Colibri telescope computers, and on Pomegranate. All scripts run on Colibri data work within this structure.

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*.

1.2 Required Modules

- sep
- numpy
- numba
- sys
- fits from astropy.io
- convolve_fft, RickerWavelet1DKernel from astropy.convolution
- Time from astropy.time
- deepcopy from copy
- pathlib
- multiprocessing
- datetime
- os
- gc
- time

2 Algorithm

The following is a basic overview of the steps the Colibri Main Pipeline takes to reduce the data. For a more detailed description of each function, see Section 3.

1. Parameters for running the pipeline are defined.

   RCDFiles (boolean) is set to *True* if the pipeline should run on .rcd images. If set to *False* the pipeline will run on .fits images.

   runPar (boolean) is set to *True* to run the pipeline in parallel mode, which will process minute-long datasets on different cores.

   The name of the telescope, the gain (*high* or *low*) for .rcd files, the date of observations, and the base path to the observation and archived data is set here.

2. Filepaths to the directories where the data is stored are defined. Note that filepaths in the pipeline are managed using python's **pathlib** library. This is so the pipeline can be run on both Linux and Windows operating systems, and to make file structure navigating and accessing more straightforward.

3. Bias images are prepared using the **makeBiasSet** function (Section 3.14). This creates a set of median combined "master bias images" to be subtracted from the science images before they are run through the pipeline. During data collection, a set of 50 bias images are taken every 15 minutes to mediate the effects of changing bias levels throughout the night.

4. The Ricker wavelet ("Mexican Hat") kernel is prepared using **RickerWavelet1DKernel** module in **astropy.convolution**. The width of this kernel is determined by the exposure time of an image and the characteristic scale length of a KBO occultation. The kernel will later be convolved with individual star light curves to search for occultation patterns in the data (Section 3.4).

5. The main part of the pipeline is run for each minute-long data set. If running in parallel, the pipeline runs on minute-long directories simultaneously on multiple cores. Otherwise, the directories are run in sequence. If the pipeline is running on .fits files, there is a check in place to see if the conversion has taken place. If not, the .rcd files are converted to .fits files before calling **firstOccSearch**. The function **firstOccSeach** is really the main function of the pipeline which runs through all the steps necessary to process the data and search for occultation events. It runs on one minute-long data directory at a time. See Section 3.5 for a detailed break down of each step.

5.1 The appropriate master bias image for the current minute is selected.

5.2 Star positional data is created from a median combined image stack.

5.3 The average drift rate of the star field over the minute is calculated.

5.4 A time series for each star over the minute-long data set is created, accounting for star field drift if necessary.

5.5 The time series of each star is checked for candidate occultation events. This is done by sending each star's time series to the **dipDetection** function (Section 3.4). This function runs through the following steps:

5.5.1 The light curve is checked for tracking failures and signal-to-noise ratio.

5.5.2 The light curve is checked for *geometric* occultation events.

5.5.3 If no events of this type are found, the light curve is convolved with the Ricker Wavelet kernel created in Step 4, and checked for *diffraction* events.

5.6 Light curves containing events are archived to be run through the secondary pipeline.

6. Steps 1-5 are completed for each minute-long dataset until the entire night is processed. When the pipeline has finished, a directory labelled by the processing date in *ColibriArchive* will contain .npy files with star coordinates for each minute, any master bias images created, median combined images for star detection in each minute, and all the detection .txt files.

3 Functions

3.1 averageDrift

- Determines the median x and y drift rates (px/s) of all stars across a minute-long data set.
- Compares the star positions between two frames, and uses the time difference between these frames to determine the drift rate.

Input:

1. 3D array of [x,y] star positions - [# frames (should be 2), # stars, # position columns (should be 2 - X & Y)]
2. list of header times of each position

Returns:

1. median x drift rate [px/s]
2. median y drift rate [px/s]

Algorithm:

1. Convert header times to unix times (for simple subtraction).
2. Get number of pixels each star has drifted in x and y separately by subtracting the last frame positions from the first frame positions.
3. Get the time interval between the first and last frame.
4. Calculate a drift rate for each star by dividing the amount of drifted pixels by the time. Take the median in x and y.

3.2 chooseBias

- Choose the correct master bias by comparing time to the observation time

Input:

1. Filepath to the directory for the current minute-long dataset.
2. 2D numpy array of [master bias image datetimes, master bias image filepaths]

Returns:

1. Bias image array that is closest in time to observation

Algorithm:

1. Get a datetime object for the current directory using **getDateTime** (Section 3.7).
2. Make an array containing the differences between each master bias datetime and the current datetime. Select the array index with the smallest difference. This is the master bias image that is nearest in time to the time the current directory was created.
3. Load in the image data for the selected master bias image and return this image array.

3.3 clipCutStars

- Originally written by Emily Pass
- When the photometry aperture is near the edge of the field of view sets flux to zero to prevent fadeout

Input:

1. List of X coordinates for each star in pixels
2. List of Y coordinates for each star in pixels
3. Length of image in x direction
4. Length of image in y direction

Returns:

1. Indices of stars to remove

Algorithm:

1. Set a threshold for the number of pixels within the edge of the image to remove stars
2. Make arrays of star x and y coordinates.
3. Get indices of stars which have their x coordinates outside the threshold on the right side of the image and on the left side. Get indices of stars which have their y coordinates outside the threshold on the top and the bottom.

3.4 dipDetection

- Modified from function written by Emily Pass and Tristan Mills
- Checks the light curve of a star for large ('geometric') dips in flux, and for smaller ('diffraction') dips in flux.

Input:

1. Light curve of a star over the minute-long data set as an array of fluxes.
2. Ricker wavelet kernel as an array of normalized counts.
3. ID number of the star.

Returns:

1. Frame number of the detected event. If no event detected, returns an error code:
   - *-1*: No events in the minute that passed detection thresholds.
   - *-2*: Light curve was too short, star was not tracked properly, SNR below the threshold, or the event was too close to the beginning/end of the series.
2. Light curve of the event (**not** normalized). Returns empty list if no detection.
3. Keyword indicating the type of event detected.
   - *Geometric*: Event is below 40% of the normalized median flux level.
   - *Diffraction*: Event is below 3.75σ of the light curve convolved with the Ricker wavelet kernel.

   Returns an empty string if no event detected.

Algorithm:

1. Prune light curves. Any leading or trailing zeroes are removed from the array. If, after this is done, the light curve is empty, it returns an error code indicating this.
2. Checks are performed to test if the light curve is good enough to look for dip events. Sets the ideal number of images per minute (2400 at 40 Hz). Sets a minimum Signal to Noise threshold, where SNR is defined as the light curve median divided by the standard deviation. Sets a minimum number of array elements for the light curve.
   - If the lightcurve has fewer elements than the minimum number the star is rejected.
   - If the difference in the star's mean flux between the beginning and the end of the minute is greater than the standard deviation, this indicates that the star has not been tracked properly. The star is rejected.
   - If the star's SNR (median/standard deviation) is below the limit set above, the star is rejected.

   There is an option here to uncomment a line that will return the star's light curve without checking for dips. This was used for debugging.
3. The light curve is convolved with the Ricker wavelet kernel using the function **convolve_fft** from astropy. The index of the minimum value in the convolution is found. This value of this minimum is found.
4. The light curve is checked for large (*geometric*) dips. The threshold for a geometric dip (*geoDip*) is set Note that this will look for a dip that is 1.00 - threshold (ie if *geoDip* = 0.6, dips of 40% or greater will be detected). The light curve is normalized by its median value.
   If the value of the normalized light curve at the index of the convolution's minimum is less than the threshold, the dip event is returned.
5. If no geometric dip has been detected, the light curve is checked for smaller (*diffraction*) dips. If the location of the convolution's minimum is too close to the beginning or end of the light curve, the star is rejected. Otherwise, a background zone is created. This background zone is the entire light curve excluding a buffer at the beginning and end of the series. A dip detection threshold (currently set at 3.75) is set.
   The criteria for a diffraction dip is:

   $$minimum(convolution) < mean(backgroundzone) - threshold \times \sigma \tag{1}$$

   where σ is the standard deviation of the background zone. If this criteria is satisfied, the dip event is returned. If not, the star is rejected.

3.5 FirstOccSearch

- Modified from a function written by Emily Pass (originally called ”Main”)
- Operates as the ’main’ function of the pipeline, calling all processing steps for each *minute* directory.

Input:

1. Filepath to folder to process [Path object]
2. List of master biases filepaths and their times [2D array of Path objects and strings]
3. Ricker wavelet (formerly known as ”Mexican Hat”) kernel [list of normalized fluxes]
4. Exposure time for a single image
5. Gain level for images [string - either ”high” or ”low”]

Returns:

1. Saves .npy file with star positions
2. Saves .txt file for each detected event

Algorithm:

1. The folder in which results are saved is created if it doesn’t exist already.
2. The median combined bias image that is closest in time to the current minute is selected using the function **chooseBias** (Section 3.2).
3. The detection threshold for star finding and the circular aperture radius for photometry are chosen.
4. A list of image filepaths within the current **minuteDir** is made.
5. The dimensions of the images and the number of images in the directory is determined using either **getSizeFITS** (Section 3.8) or **getSizeRCD** (Section 3.9). The current minute directory is skipped if there are too few images (set to 3x the kernel length).
6. Star positional data is created from a median combined stack. Image stacking is performed using the **stackImages** function (Section 3.20). This image is passed to **initialFind** (Section 3.12) to perform object detection using the **sep.extract** module. The object positions and half-light radii are saved in a .npy file labeled ***minuteDirName_detectThresh****sig_pos.npy* in the results directory. If there are too few stars found, this step will be repeated until enough detections are made.
7. The average drift rate of the star field over the minute is calculated. This is done by importing the first and last images in the set using either **importFramesFITS** (Section 3.10) or **importFramesRCD** (Section 3.11). Each of these images are passed to **refineCentroid** (Section 3.17) along with the initial star positions to refine the positions using **sep.winpos**. The average drift rate of stars over the minute-long data set is calculated using **averageDrift** (Section 3.1). If the average drift in either the x or y direction is greater than the threshold, drift will be accounted for when doing photometry.
8. A time series for each star over the minute-long data set is created, accounting for star field drift if necessary. For each image in the series, the image file is imported using the function **importFramesFITS** (Section 3.10) or **importFramesRCD** (Section 3.11). This returns the image array and the header time. This information along with the previous image’s star positions is passed to **timeEvolve** (Section 3.22) or **timeEvolveNoDrift** (Section 3.23) if the drift is significant. These functions perform aperture photometry using the **sep.sum_circle** module, including local background subtraction in an annulus. The output of this step is a multidimensional array with dimensions [frames, star number, X coord, Y coord, list of fluxes, list of unix times].
9. The time series of each star is checked for candidate occultation events. This is done by sending each star’s time series to the **dipDetection** function (Section 3.4). This function returns the original light curve and the frame at which the dip is found, and the type of event (*diffraction* or *geometric*).

10. Light curves containing events are archived to be run through the secondary pipeline. A list of frame numbers where events were detected is created from the results of the dip detection step. A list of light curves containing events is also created. The number of frames on either side of the detected event to archive is calculated as well.

    For each detected event a .txt file is created that saves the important data about the event. This file has a header with the name of the image where the event occured, the header date and time for the event, the star coordinates, telescope name, and star field identifier. It also contains a table with the list of image filepaths, the image time (seconds only), and the star flux.

11. Printout statements with the number of detected stars and percentage of events are printed to the screen.

3.6 getBias

- Make a median combined bias image to use for bias subtraction.

Input:

1. Filepath to directory containing bias images [pathlib path]
2. Number of bias images to include in the median combine [int]
3. Gain ('high' or 'low') of .rcd or .fits images. [string: 'high' or 'low']

Returns:

1. Median combined bias image [2D array]

Algorithm:

For .fits files:

1. Create a variable with the name of the 'check file'. This is an empty file that indicates whether a .fits conversion has been done to each of the .rcd files in the directory.
2. If this file doesn't exist, the .fits to .rcd conversion script (**RCDtoFTS.py**) is called to do the conversion with the specified gain. If the file does exist, the files are already in .fits format and the script can proceed.
3. Make a list of all .fits files in the directory. Prepare an empty list to hold data for these images.
4. For each image in the list of files, read in the data and add the image array to the bias list.
5. Take the median of these images to create a new median combined image of the biases. Return this.

For .rcd files:

1. Make a list of all .rcd files in the directory.
2. Import each .rcd image in the list using the function **importFramesRCD** (Section 3.11). Pass it the list of bias images, the starting index (set to 0), the number of biases to import, and empty array of the same dimensions to serve as the 'bias' for these images, the gain value. These imported images are returned as a list.
3. Take the median of the images in the list to create a new median combined image of the biases. Return this.

3.7 getDateTime

- Get a datetime object from the name of a data directory using the python module **datetime**. Datetime objects can be manipulated and compared more easily than basic strings when doing math that involves dates and times.
- Each minute long dataset is contained in a directory labelled *'yyyymmdd_hh.mm.ss.uuu'*. This function takes this string and converts it into a datetime object *'yyyy-mm-dd'* and *'hh:mm:ss:uuu'*

Input:

1. Filepath to minute-long dataset directory. [path object]

Returns:

1. Date and time contained in the directory name [Datetime object].

Algorithm:

1. Separate the part of the directory name string that contains the time.
2. The date is taken using the global *obs_date* variable set at the beginning of the main script.
3. A **datetime** *time* object is created from the directory time.
4. This is combined with the directory date to create a **datetime** *datetime* object.

3.8 getSizeFITS

- Originally written by Emily Pass and Tristan Mills
- Get the number of .fits images in a data directory (for minute-long data sets should be around 2400). Get the dimensions of .fits images.

Input:

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

Returns:

1. width of .fits image [px]
2. height of .fits image [px]
3. number of .fits 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.

3.9 getSizeRCD

- Modified from functions written by Emily Pass and Tristan Mills
- 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.

3.10 importFramesFITS

- Based off functions written by Emily Pass and Tristan Mills
- Reads in multiple images from .fits files in the directory

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.

Returns:

1. List of image data arrays [list of floats]
2. Array of header times for these images [1d array of strings]

Algorithm:

1. Create empty lists to hold the image arrays and header times.
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.
    3.1 Open the image using the **fits** module in **astropy**. Read in the image header. Read in the image array (counts) and subtract off the bias image. Get the timestamp of the image.
    3.2 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.
    3.3 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.

3.11 importFramesRCD

- Based off functions written by Emily Pass and Tristan Mills. Adapted from **importFramesFITS** (Section 3.10) by Mike Mazur.
- Reads in multiple images from .rcd files in the directory.
-

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]

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.
   - 3.1 Open the image using the **readRCD** (Section **??**), which returns both pixel data and header. Get the timestamp of the image.
   - 3.2 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 **??**). Subtract the bias image data from the image data.
   - 3.3 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.
   - 3.4 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.

3.12 initialFind

- Based off a function written by Emily Pass
- Locates stars in an image.

Input:

1. Flux data for an image [2D array]
2. Detection threshold coefficient for star finding

Returns:

1. Zip object for all objects containing X coordinate [px], Y coordinate [px], and half light radius [px].

Algorithm:

1. Extract the background for the image. This is done using functions in the **sep** module.

   First, a copy of the data is made. Then, a 2D array of the variable background across the image is extracted using the **sep** class **Background**.

   This background array is subtracted from the image array copy using the **sep.Background** method **subfrom**.

   The significance threshold for star detection is set. This is determined by multiplying the given detection threshold level with the global root-mean-square of the background array (using the **sep.Background** attribute **globalrms**.
2. Stars are identified in the background subtracted image. This is done using the **sep** funcition **extract**. The significance threshold calculated above is passed to the function as the pixel threshold value for detection. The object returned from this extraction is a structured array. See **sep.extract** documentation for more details.
3. The star's half-light radius is approximated as half the star's radius as returned by **sep.extract**.
4. A zip object containing the X, Y coordinates and half light radius for each star is generated.

3.13 nb_read_data

- Written by Mike Mazur.
- Function to read 12-bit data from .rcd files
- Uses numba to speed things up.

Input:

1. Chunk of 12-bit data to be read in.

Returns:

- Pixel data from the image (fluxes/counts).

3.14 makeBiasSet

- Get set of median-combined biases for entire night that are sorted and indexed by time. These are saved to disk and loaded in when needed.

Input:

1. Filepath to bias directories [pathlib Path object]
2. Number of biases to combine for master [int]
3. Gain keyword ['high' or 'low']

Returns:

1. 2D array with master bias timestamps and filepaths.

Algorithm:

1. Make list of bias minute folders and make a folder to save the results in (See Figure 2).

2. Loop through each bias minute folder.
    2.1 Make the median combined bias image using **getBias** (Section 3.6).
    2.2 If the master bias image doesn't already exist on the disk, save it as a .fits file.
    2.3 Get the time that the bias minute folder was created from the folder name. This is done using **getDateTime** (Section 3.7).
3. Package all times and filepaths into a single array, sorted by time.

3.15 readRCD

- Written by Mike Mazur
- Reads .rcd image and returns image data (pixel array and header)

Input:

1. Path to the image file [string or path object]

Returns:

1. 2D numpy array of pixel data
2. Dictionary of header data with keys 'serialnum' and 'timestamp'

Algorithm:

1. Make empty dictionary to hold image data.
2. Go to start of file.
3. Read in the bytes corresponding to the camera's serial number.
4. Read in the bytes corresponding to the timestamp.
5. Read in the bytes corresponding to the image data.

3.16 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]

3.17 refineCentroid

- Originally written by Emily Pass
- Refines each stars centroid based on the image and its previous coords
- Note that this function is designed for cases where there has not been significant drift. If the star has drifted so far from its original location that it isn't within its original aperture, the function will fail. For these cases, another method must be used.

Input:

1. Flux data in 2D array for single image
2. Header time of image
3. list of star coordinates in previous image
4. weighting factor (Gaussian sigma)

Returns:

1. New [X,Y] positions (tuple)
2. header time of image

Algorithm:

1. Separate out lists of X and Y coordinates.
2. Use the **sep.winpos** function to make an array of new [x,y] position sets. This function uses an iterative windowed algorithm to make a better estimate of each star's centroid. The coordinates are separated out again into separate lists.

3.18 runParallel

- Written by Mike Mazur
- Call the 'main function' (**FirstOccSearch**, Section 3.5) with the desired arguments. Meant for running in parallel.

Input:

1. The arguments passed to **FirstOccSearch** (Section 3.5).

3.19 split_images

- Written by Mike Mazur.
- Extract either high gain or low gain image from .rcd file.

Input:

1. Data from .rcd file.
2. Image height [px]
3. Image width [px]
4. Image gain level (string, 'high' or 'low')

Returns:

1. Image data from desired gain level.

3.20 stackImages

- Make median combined image stack for star finding.

Input:

1. Directory of images [Path object]
2. Directory to save stacked .fits image to [Path object]
3. Starting index to begin stack [int]
4. Number of images to combine [int]

5. Bias image to subtract from stack [2D array]
6. Gain level (string, 'low' or 'high')

Returns:

1. Median combined stack of bias-subtracted images

Algorithm:

1. For processing .fits files:
   - 1.1 A list of all .fits files in the directory is made, sorted by image index.
   - 1.2 Each individual image array is appended to a list after being imported using **importFramesFITS** and bias-subtracted (Section 3.10).
   - 1.3 A new image array is created from the median of the .fits files.
   - 1.4 A new header object is created to save the data to.
2. For processing .rcd files:
   - 2.1 A list of all .rcd files in the directory is made, sorted by image index.
   - 2.2 Each individual image array is appended to a list after being imported using **importFramesRCD** and bias-subtracted (Section 3.11).
   - 2.3 A new image array is created from the median values of the .rcd files.
   - 2.4 A .fits header object is created to save the data to.
3. A filepath for the median image is created. The image will be saved in the input save directory, and will be labelled as ***'gain_yyyymmdd_hhmmss.mmm***_*medianstacked.fits*.
4. If the image does not already exist, it is saved to this location.

3.21 sumFlux

- Note that this function is not used in the current iteration of the main pipeline. It was written to see if this method would be faster than **sep.sum_circle**, which it was not.
- Function to sum up flux in square aperture of size: $(2xlength + 1)^2$

Input:

1. image data [2D array]
2. list of X coordinates for all stars
3. list of Y coordinates for all stars
4. half side length of box [px]

Returns:

1. List of fluxes for each star [pixel counts]

Algorithm:

1. Make a list of image pixel values in the region around each star. This is done by making a list of pixel values for each star within the box. The box is defined as the star coordinate +/- the input box length.
2. Make a list of star fluxes by summing the pixel values in the region for each star. The final product is a list of fluxes over time for each star.

3.22 timeEvolve

- Modified from an original version by Emily Pass
- Adjusts each star's aperture accounting for drift, gets the flux for that star in a single image

Input:

1. Image data [2D array of fluxes]
2. Header time for image
3. Star data from previous image (coords, flux, time of image)
4. X direction drift rate [px/sec]
5. Y direction drift rate [px/sec]
6. Aperture radius to sum flux in [px]
7. Total number of stars
8. Image X width [px]
9. Image Y height [px]

Returns:

1. Tuple of zip objects for each star containing star's new x coord, new y coord, flux, frame time

Algorithm:

1. Get the header time for the current frame and convert it to a **Time** object. Find the amount of time elapsed since the previous frame for the drift calculation.
2. Adjust each star's positions to account for drift. This is done by multiplying the X and Y drift rates by the number of seconds elapsed since the previous frame. Save the X and Y positions in 2 separate lists.
3. Make a list of stars which are close to the edge of the frame and may drift out of frame. Make a new list of stars with these removed.
4. For each star, sum up the flux in the aperture. In the current iteration of the pipeline (July 2022) this is done using **sep.sum_circle**. A previous version used the function **sum_flux** (Section 3.21), but this was found to be slower than using **sep**.

   We ask **sep** to do background subtraction in an annulus around the star. The radii for this annulus was found empirically (6 - 11 px as of July 2022). The final product of this function is a list of star fluxes in the order of the star list.
5. Any stars that have drifted out of frame are given a flux value of 0 in this list. This is done in part to keep indices consistent and to track what happened to all stars detected in the initial star finding step.
6. All star data is packaged in a tuple to be returned, unpacked, and reformatted into the results array.

3.23 timeEvolveNoDrift

- Modified from an original version by Emily Pass
- This function is essentially the same as **timeEvolve** (Section 3.22), but without accounting for drift.
- Gets the flux value for each star in the current image

Input:

1. Image data [2D array of fluxes]
2. Header time for image

3. Star data from previous image (coords, flux, time of image)
4. Aperture radius to sum flux in [px]
5. Total number of stars
6. Image X width [px]
7. Image Y height [px]

Returns:

1. Tuple of zip objects for each star containing star's new x coord, new y coord, flux, frame time

Algorithm:

1. Get the header time for the current frame and convert it to a **Time** object.
2. Make a list of all x positions and a list of all y positions.
3. Make a list of stars which are close to the edge of the frame and may drift out of frame. Make a new list of stars with these removed.
4. For each star, sum up the flux in the aperture. In the current iteration of the pipeline (July 2022) this is done using **sep.sum_circle**. A previous version used the function **sum_flux** (Section 3.21), but this was found to be slower than using **sep**.

   We ask **sep** to do background subtraction in an annulus around the star. The radii for this annulus was found empirically (6 - 11 px as of July 2022). The final product of this function is a list of star fluxes in the order of the star list.
5. Any stars that have drifted out of frame are given a flux value of 0 in this list. This is done in part to keep indices consistent and to track what happened to all stars detected in the initial star finding step.
6. All star data is packaged in a tuple to be returned, unpacked, and reformatted into the results array.