AstroPyplyne: ARCSAT Data Reduction in Astropy

Taylor Fleming

A senior thesis submitted to the faculty of
Brigham Young University
in partial fulfillment of the requirements for the degree of

Bachelor of Science

Denise Stephens, Advisor

Department of Physics and Astronomy

Brigham Young University

ABSTRACT

AstroPyplyne: ARCSAT Data Reduction in Astropy

Taylor Fleming
Department of Physics and Astronomy, BYU
Bachelor of Science

IRAF is a commonly-used software for data reduction but movement is being made toward Python-based methods. I developed an Astropy pipeline, AstroPypelyne, starting with the Astropy workshop from the 241st meeting of the AAS. AstroPypelyne receives raw data from the ARCSAT telescope, performs reductions and corrections, and distributes the processed files into newly created folders by image type and object. Calibration frame selection is automated and files that cannot be reduced will be ignored and the user informed. Cosmic rays and hot pixels are optionally removed during the image processing sequence. Data reduction using this pipeline is faster and more efficient than IRAF methods yielding results of the same caliber. Once fully polished, AstroPypelyne will be distributed and modified for other optical telescopes.

Keywords: ARCSAT, Astropy, Data Reduction, Data Processing

ACKNOWLEDGMENTS

Thanks to Mr. Jay Eads, for piquing my interest in astronomy, Dr. Darin Ragozzine for pointing me in the right direction, and Dr. Denise Stephens for helping me give back to the department. Additional thanks to my research group for helping me to overcome hurdles in the code and my wife Audrey for supporting me through everything.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

To discover planets orbiting other stars called exoplanets, make multiple observations of target stars over the space of a night. Light-curve analysis compares the light received across these observations and seeks dips in brightness from transiting exoplanets. These light counts are measured using charge-coupled devices (CCD), doped semiconductor sensors that record received light by counting electrons excited by photons. The images received from telescope observations are unusable for light-curve analysis until they have been processed. Light counts recorded in these images are muddled by light and noise from the sky and atmosphere in addition to telescope noise and imperfections. Data processing, also called data reduction, uses calibration frames to clean up data for analysis.

1.1 Calibration Frames

Calibration frames come in three types: bias, dark, and flat. Bias frames represent the inherent flux in the chip measuring light counts. This electric flux is the current run through the camera's chip to avoid negative and zero inherent readings. These are taken by making snapshot observations with the telescope, only collecting light and noise readings for an instant. Dark frames show counts

measured resulting from electrons jumping into the conduction band of the CCD. These are taken over several exposure times to match the integration times from observing target stars. Flat frames correct for uneven illumination of the CCD as edges will receive less light than the center. They also account for differences in pixels' sensitivity to light. They're created by observing either the telescope dome or a dim patch of sky with relatively constant brightness. These frames are taken in each light filter used to observe stars that night.

We use these calibration frames first to correct one another before applying them to our images. The bias frames are first combined by averaging them into a super bias. Using this super bias, we correct all our dark frames by removing the noise inherent to observation that our super bias represents. We then combine these modified dark frames into super darks separated by exposure time giving us a super dark per unique exposure time. Flat frames are corrected by the super bias in the same way as the dark frames. They then have the telescope noise represented in the super darks removed from them using the super dark that best matches each flat frame's exposure time. In the rest of this paper, we will refer to super calibration frames as a super bias, super dark, or super flat when discussing these combined calibration frames.

1.2 Reducing Telescope Data

Using each of these super calibration frames, we process our images. For each image, we begin by subtracting the inherent noise out using the super bias. Then, matching the image's observation time to a corresponding super dark's exposure time, we remove the additional telescope noise reflected in extra counts. Finally, we match the filter used to obtain each image with the associated super flat and correct the image for inconsistencies in pixel sensitivity and CCD illumination. Software is readily available to perform this process, but it requires the user to run each step.

1.3 The Depreciation of IRAF

Image Reduction and Analysis Facility, or IRAF, is one such software whose many capabilities established it a common means for data reduction (Tody 1986). It's able to reduce and correct data sufficient for analysis. However, it is no longer supported, and alternatives are phasing in. IRAF requires a lot of effort, which bottlenecks the flow of research when large batches of telescope images come in from a series of consecutive nights. The user must manually combine all the calibration frames and correct the images with the corresponding super calibration frames. With so many stages in the process, data processing becomes cumbersome and holds research back.

1.4 An Astropy Pipeline

Astropy is IRAF's worthy successor (Astropy Collaboration et al. 2013; 2018; 2022). Astropy has many packages that mirror IRAF while including potential to be automated. Functions not natively available may be written in and incorporated. Astropy optimizes the reduction process through trimming user input and its faster processing speeds. This minimizes the time between observation and analysis.

These advantages drove the development of AstroPypelyne, an Astropy pipeline written to relegate data processing to the background. It produces analysis-ready images when given raw data and calibration frames with only limited input required. While written specifically for data from the ARCSAT telescope at Apache Point Observatory (APO), its capabilities may be extended to use with other optical telescopes with some modification of keywords and processes.

In Chapter 2, we discuss my development of AstroPypelyne. In Chapter 3, we compare AstroPypelyne with IRAF methods in efficiency and results. In Chapter 4, we consider what comes next for AstroPypelyne. In Appendix A, a README file is included, outlining the function of each cell in the code, explaining its process and adaptability.

Chapter 2

Methods - Developing the Pipeline

I began development on AstroPypelyne by following the "Using Python and Astropy for Astronomical Data Analysis" workshop (accessed via GitHub) from the 241st Meeting of the American Astronomical Society (AAS). This workshop, in conjunction with the CCD Data Reduction Guide by Matt Craig and Lauren Chambers (Astropy Collaboration et al. 2022), formed the backbone for AstroPypelyne. These resources walk through setting up Astropy, importing necessary packages, and reducing telescope data. These packages, shown in the Table 2.1 below, include ccdproc (Craig et al. 2017), numpy (Harris et al. 2020), and photutils (Bradley et al. 2023), with Astro-SCRAPPY, the implementation of L.A. Cosmic, (McCully et al. 2018; van Dokkum 2001).

ccdproc	CCD image processing functions that apply super calibration frames to images
numpy	Mathematical operations and functions used for statistics
photutils	Functions to identify stellar sources when masking them out
Astro-SCRAPPY	Corrects for cosmic rays that appear very bright in individual images

Table 2.1 Brief description of functions imported according to the CCD Data Reduction Guide.

2.1 Basic Reduction

I built AstroPypelyne in a Jupyter Notebook, separating out the steps of the data reduction process into individual cells. In each step, I imported the necessary images or calibration frames, assigned units to the files, and processed the data following the steps briefly outlined in Section 1.2. The guide and workshop noted above outline this process in Astropy. Since each object image was taken by the telescope using a light filter and for a set integration time, I sorted object images and calibration frames by filter and exposure time so they would be correctly calibrated during reduction.

2.2 Automated Header Identification

To cut this need for the user to sort the dataset in this way before performing data reduction, I incorporated checks that allowed AstroPypelyne to read information from the header files of each image. These header files contain information including type (light, dark, flat, bias), filter used, and exposure time. By having the code identify this information automatically, I shifted many steps in the process from the user's hands to AstroPypelyne.

I implemented this capability by including the identifier for the desired image type each time data is imported. AstroPypelyne distinguishes between these attributes in dark and flat type images. It generates lists containing all unique filters and exposure times found in the dataset. It then iterates through these lists at each instance of dark or flat combination or final image reduction. AstroPypelyne matches the correct super calibration frame to each image being reduced. This capability was a major step forward in developing AstroPypelyne as I had a Jupyter notebook that could accomplish all the steps we perform in IRAF on its own.

2.3 Organization of Output

An issue remained, however, that kept AstroPypelyne from being rerun without replacing input images: corrected images were replacing the original files. I remedied this by listing the identifiers of the objects observed in the data set and creating new folders where each object's images would be saved. Because the object name is found in ARCSAT's image file names, they were read into lists as with the filters and exposure times to identify all unique objects. I paired an abbreviated identifier with the full name using Python dictionaries to expand these lists. I coded AstroPypelyne to iterate through these lists to create output folders for each observed object. The generated folders' names were identical to the object's abbreviated identifier. At the end of the process, each reduced image was saved to the folder created for its object. Calibration frames and super calibration frames likewise were saved to new folders to avoid overwriting original data. In Figure 2.1 below, the folder structure is shown to help visualize the benefit of this change. Only one folder must be created by the user while all others are generated (and refreshed when AstroPypelyne is run again) automatically.

2.4 Further Corrections

At this stage for AstroPypelyne, data reduction could be performed with no file sorting required of the user. Output was sorted and code-generated folders refreshed upon the pipeline being rerun. With basic reduction covered, I added more corrective procedures including bad pixel blurring and cosmic ray corrections.

2.4.1 Bad Pixel Blurring

Bad pixel blurring is a two-step process performed at several steps in the process. I first identified those pixels that are not accurately measuring light, identifying them by their extreme values, then

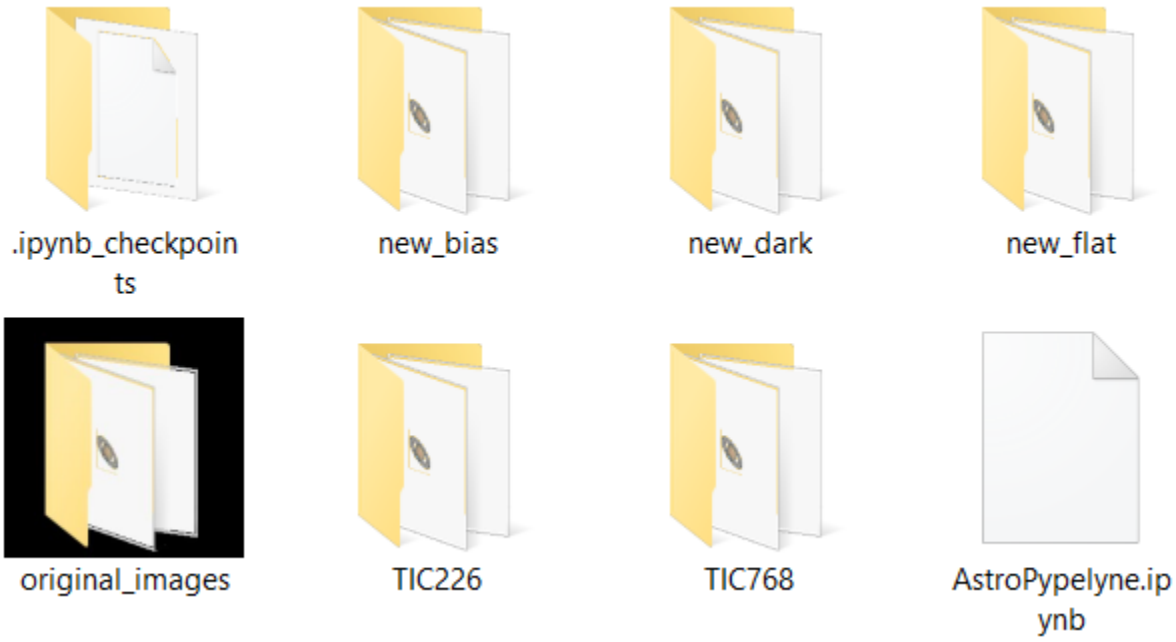

Figure 2.1 A visual showing the organization of folders with data on Windows. The folder titled original_images and the notebook are the only ones which the user must first make while the rest are automatically generated by AstroPypelyne. Each of these separate folders contributes to avoiding overwriting files and sorts the output for the user's convenience. The objects observed in this example dataset have identifiers TIC22683951 and TIC76882795 which have been abbreviated for visual convenience.

blended them with their surroundings. This must be performed on all images and calibration frames as the bad pixels will exist for every image from the CCD. I simplified correcting this discrepancy in calibration frames by making these corrections on just the super calibration frames. The rest of the images must be processed individually, which I integrated into the other reduction steps.

To identify bad or "hot" pixels, I examined super darks with the highest and lowest exposure times. Comparing counts per second of exposure, I found a ratio identifying the standard amount of additional counts on the image versus the amount of time spent making the "observation." Those pixels that do not fit this trend within a user-defined range were flagged as bad pixels.

I corrected the bad pixels using a simple function I defined in the pipeline. Each bad pixel is given a new value equal to the average of the pixels directly around it to blend it in with surrounding pixels above, below, and on either side. Bad pixels on the edge of the image can also be corrected as the function I wrote only considers those neighboring pixels which lie within the range of the image. For bad pixels on the edges of the image, no pixels are sought beyond the image's bounds and the sum is divided by one less pixel per edge.

2.4.2 Cosmic Ray Corrections

Cosmic ray corrections followed a similar process, but utilized an existing package built into ccdproc. This implementation of L.A. Cosmic by (van Dokkum 2001) and (McCully et al. 2018) does a lot of the work for us, identifying cosmic rays in each image and performing the necessary corrections. While these aren't perfect with our data, the user may adjust the rigor and scope of cosmic ray identification to clean up the images without stars being confused for cosmic rays.

For both bad pixel and cosmic ray corrections, I included variables defined by boolean values in the third cell of AstroPypelyne that the user can toggle. This way, the degree of correction performed on data is left in their hands. The user must simply change a True to a False or vice-versa to specify what additional corrections to perform.

Chapter 3

Results - Comparing the Processes

Once AstroPypelyne proved capable of full data reduction, we compared it to IRAF in terms of required user input, execution speed, and output image quality.

IRAF requires the user to run each step of the process, resulting in several instances of user input. Before this process begins, the images and calibration frames must be sorted. All of this sums to a large amount of user involvement in the process of reducing data per night of observation.

Getting started with AstroPypelyne, the user inputs a file path to identify the location of telescope images. This initial step is the only necessary user input for most cases as all sorting by image type, exposure time, and filter is conducted internally. After first execution, AstroPypelyne requires no additional user input to correct images from other datasets. The folder containing the uncorrected images must only be emptied and replaced with new data. This folder acts as a hopper for data to be fed into the pipeline and processed. If super darks cannot be found to fit images or flats, AstroPypelyne notifies the user and they may supplement the raw data with the necessary images or adjust the number of seconds within which exposure times must match. This number is set in AstroPypelyne as a variable named 'tolerance' in the same cells as these checks occur.

When all user input is taken into account, using AstroPypelyne is significantly faster than IRAF. AstroPypelyne handles batches of hundreds of images in around a minute. For an experienced user

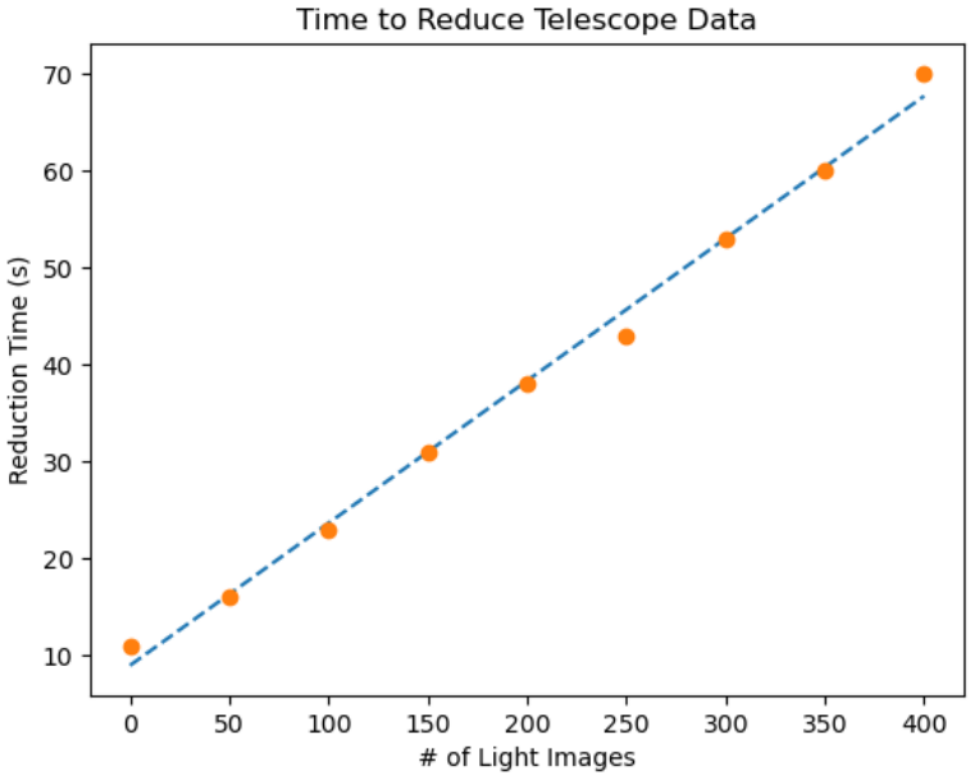

Figure 3.1 This graph shows a linear relation between the number of images to be reduced and the time for AstroPypelyne to process the images. The measured times represent the time between importing functions and AstroPypelyne finishing reduction.

of IRAF, most image batches take 10-15 minutes to process. In addition to trimming delays from the user input, each step runs more quickly using AstroPypelyne as Astropy iterates through the steps more efficiently than IRAF's engine. In Figure 3.1, the data points show recorded times of AstroPypelyne processing the same data with varying numbers of images. Along with these points is a line of best fit showing a linear relation between number of images to reduce and reduction time. This linear fit indicates that calibration frames take approximately 9 seconds to run with an additional 0.147 seconds per image to be reduced. While not shown, IRAF reductions follow a more constant relationship, taking about the same amount of time for most batch sizes.

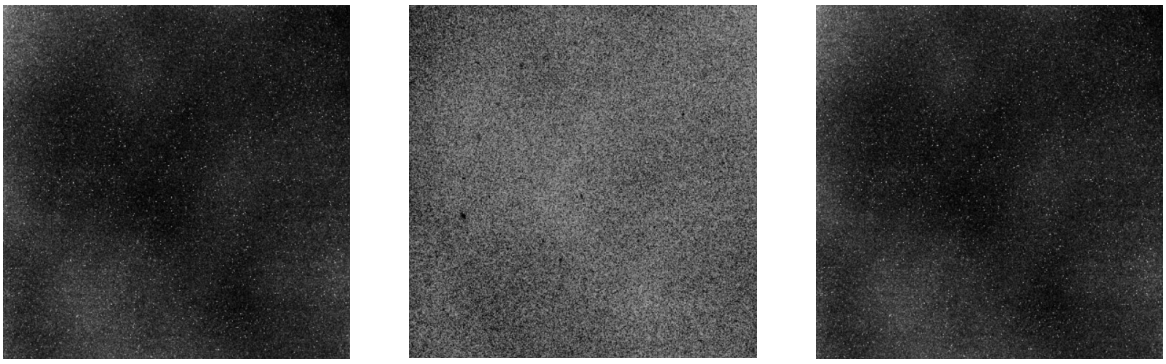

Figure 3.2 These images are super dark calibration frames with 180 second exposure times. One was produced using IRAF (left) with the other by AstroPypelyne (right). The middle image is the result of one image subtracted from the other. The IRAF-reduced super dark has a mean value of 374.88, AstroPypelyne has 377.90, and on average they differ by 3.02.

3.1 Quality of Results

The output of these two processes is visually indistinguishable for all datasets reduced by both processes. We compared the super calibration frames and randomly selected images of each object for a given night and found no significant difference in the light measured between them. Figure 3.2 shows the super darks, Figure 3.3 shows the super flats, and Figure 3.4 and Figure 3.5 show an image each of the two objects imaged on a sample night. In the caption of each figure, notable statistics are included, demonstrating the similarity of the output.

The mean difference in the super dark images in light counts per pixel is 3.02, less than a percent of the mean number of light counts for either image. This minute difference indicates that the super darks generated by either method yield sufficiently similar results. The super flats have even more similar results with a mean difference of only 5.1×10^{-9} light counts per pixel. This difference is so small that these images are effectively identical at this level of precision. The other two sets of compared images are of two objects from the same batch of data. Their mean differences in light counts are merely 2.70 and 4.22. These values are each about a percent or less of the mean value of

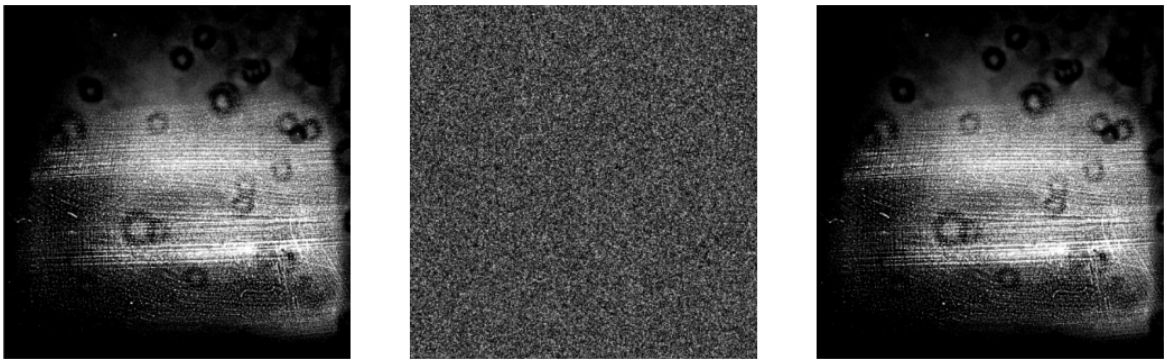

Figure 3.3 These images are super flat calibration frames in the r filter. One was produced using IRAF (left) with the other by AstroPypelyne (right). The middle image is the result of one image subtracted from the other. The IRAF-reduced super flat has a mean value of 1.0, AstroPypelyne has 0.99, and on average they differ by 5.1×10^{-9}.

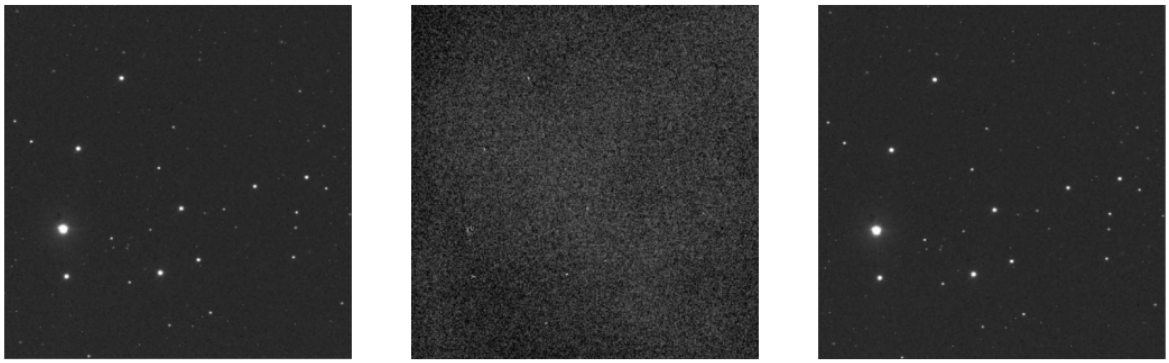

Figure 3.4 These images are sky images of the object TIC22683951. One was produced using IRAF (left) with the other by AstroPypelyne (right). The middle image is the result of one image subtracted from the other. The IRAF-reduced image has a mean value of 208.25, AstroPypelyne has 210.95, and on average they differ by 2.70.

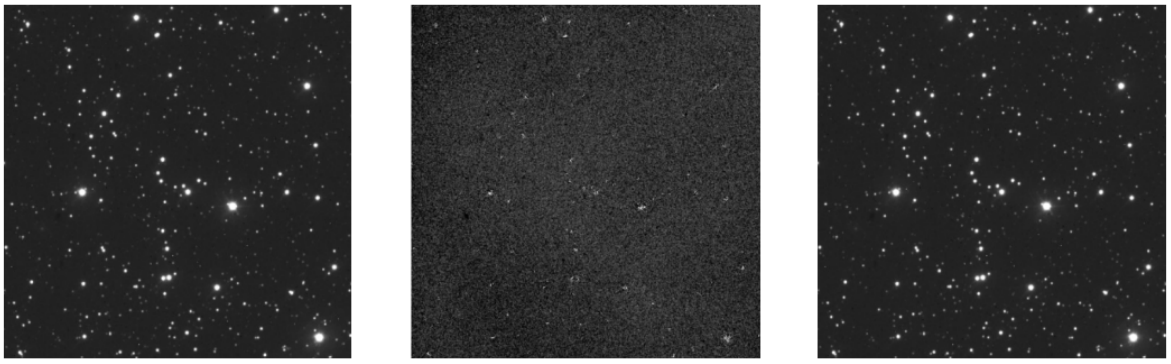

Figure 3.5 These images are sky images of the object TIC76882795. One was produced using IRAF (left) with the other by AstroPypelyne (right). The middle image is the result of one image subtracted from the other. The IRAF-reduced image has a mean value of 810.06, AstroPypelyne has 814.29, and on average they differ by 4.22.

the processed images. Across all the comparisons made, no significant differences were found in images produced by either process.

Chapter 4

Discussion - Changes to Research

AstroPypelyne expedites data reduction and significantly cuts down on delays between receiving data from the telescope and performing analysis. It outputs results of equal value more quickly and with minimal user input required. As seen in the comparisons made with IRAF, AstroPypelyne performs all functions at least as well while improving in useful ways.

ARCSAT telescope data arrives in batches every few months and using more involved methods of reduction like IRAF begin slowly, speeding up as the user is refreshed on its use. Using IRAF involves many steps that the user must perform in the correct order. After months of not being used, it's easy to forget the details of this process, requiring the user to review a guide. AstroPypelyne bypasses this issue with its accessibility; the user must only copy the images into the input folder and run the script.

Research using data from the ARCSAT telescope will be greatly bolstered by AstroPypelyne as we transition into using it in the department. New researchers will have usable data available without the prerequisite of learning data processing the long way. Even so, the script is organized in a way to clearly indicate the steps being followed. When AstroPypelyne is sufficiently polished, it will be shared with all users of ARCSAT data as a resource to reduce data. The notebook and

Python script will be posted to Github with a README file and linked to APO's website providing access to other ARCSAT users.

Another path forward for AstroPypelyne is increasing its data processing repertoire. AstroPypelyne improves on IRAF with basic data reduction, but there are many more packages in IRAF that may be used to manipulate and analyze data. Even though not all have direct equivalents in Astropy's documentation, they may yet be imitated with user-created Python functions. After creation, these functions would be usable for future analysis. The bad pixel blurring function I wrote for AstroPypelyne is an example of this as its function imitates IRAF's fixpix function to clean up bad pixels. This function is defined in an early cell in AstroPypelyne's script and used later like any other function.

One of AstroPypelyne's limitations is its specific ability to process data from the ARCSAT telescope. It was created primarily for this telescope's data, but it should be sufficiently adaptable to provide a framework to reduce other telescope's data. These other telescopes must be optical telescopes like ARCSAT, however, since the data reduction process differs for telescopes of other regimes. To adapt to other optical telescopes, keywords used to identify necessary information in each image's header file must be updated so AstroPypelyne can extract the necessary information. For ARCSAT, these keywords include 'EXPTIME' for exposure time and 'FILTER' for the filter used when observing. The first telescopes for which AstroPypelyne will be adapted will likely be BYU's own optical telescopes.

AstroPypelyne, despite its limitations, makes significant improvements on the data reduction process. It performs image reduction in noticeably less time, is almost entirely automated, and yields results of comparable quality to IRAF. With this program in use, analysis will quickly follow observation and more telescopes may become compatible, optimizing future research.

Appendix A

README Guide to Using AstroPypelyne

How to run AstroPypelyne at BYU:

- Update the file path where the notebook and *original_images* folder are located and ensure that a folder *orignal_images* containing the unprocessed data is found there.
- Activate the Astropy environment by typing *conda activate telescope* into the terminal.
- Run the python script by entering *python3 AstroPypelyne.py*.

Below is every block of code as viewed in the Jupyter notebook with an explanation of its function and any additional notes. The version shown below is AstroPypelyneDx, the version that includes the option for further corrections.

Block 1

```
# Import necessary libraries
from pathlib import Path
from astropy.io import fits
from astropy.nddata import CCDData
from photutils.detection import DAOStarFinder
from astropy import units as u
from astropy.convolution import convolve
import numpy as np
import ccdproc as ccdp
import os
import shutil
from astropy.wcs import WCS

# Supress INFO and WARNING popups while still displaying ERRORs
from astropy import log
log.setLevel('ERROR')
```

This cell is mostly composed of imports of libraries as prep for later. The final block of code prevents INFO and WARNING popups which occur all the time when the code handles .fits files. INFO and WARNING popups don't cause issues.

Block 2

```
# define paths for modified files and create new folders
filespath = 'INSERT FILE PATH HERE' # USER INPUT: Update this with location of notebook (MUST END WITH '/')
imgpath = Path(filespath + 'original_images') # USER INPUT: Update this with the folder in the above directory containing the images

# make a list of paths to save adjusted files
image_u = filespath + 'image_u'
image_u_path = Path(image_u)
new_bias = filespath + 'new_bias'
new_bias_path = Path(new_bias)
new_dark = filespath + 'new_dark'
new_dark_path = Path(new_dark)
new_flat = filespath + 'new_flat'
new_flat_path = Path(new_flat)
cosmic = filespath + 'cosmic_ray_corrected'
cosmic_path = Path(cosmic)
paths = [image_u, new_bias, new_dark, new_flat, cosmic]

for path in paths:
    if os.path.exists(path):
        shutil.rmtree(path)
    os.makedirs(path)

# NOTE: RE-RUNNING THIS CELL WILL DELETE FOLDERS OF NEWLY REDUCED DATA AND WILL FORCE YOU TO RERUN THE WHOLE NOTEBOOK
```

This cell accepts paths manually written in by the user and makes a series of folders to house the reduced data each step of the way. The bottom *for* loop checks for the folders before creating them at the newly defined paths. If there's a folder of the same name already there, the old version will be deleted and a new empty version created.

Note: *filespath* must end with a '/', otherwise paths won't be created correctly.

Block 3

```
# choose level of correction to be made for cosmic rays and bad pixels (USER INPUT: Select degree of correction)
bad_pixel_mask = True # Mask out bad pixels during data reduction
cosmic_correct = False # Cosmic ray correction (no masking before corrections)
psf_mask = False # Mask for stars before performing cosmic ray corrections

# input values for bad pixel masking
short_dark_xp = 17.0 # USER INPUT: exposure time for short dark of choice
long_dark_xp = 180.0 # USER INPUT: exposure time for long dark of choice

# bad pixel correcting function
def fix_pix(image, mask):
    top_edge = np.shape(image)[1]
    right_edge = np.shape(image)[0]
    badx,bady = np.where(mask)
    for n in range(len(badx)):
        tot_val = 0
        pix_ct = 0
        up = bady[n]+1
        down = bady[n]-1
        left = badx[n]-1
        right = badx[n]+1
        if down > 0:
            tot_val += down
            pix_ct += 1
        if left > 0:
            tot_val += left
            pix_ct += 1
        if up <= top_edge:
            tot_val += up
            pix_ct += 1
        if right <= right_edge:
            tot_val += right
             pix_ct += 1
        image.data[badx-1,bady-1] = tot_val/pix_ct

    return image

# input values for cosmic ray correction
gain = 1.25 # USER INPUT: INPUT GAIN (in electrons/adu)
read_noise = 11.8 # USER INPUT: INPUT READ NOISE (in electrons)

# input values for star identification
threshold = 5000 # USER INPUT: Input minimum number of counts to identify as a star
FWHM = 2.5 # USER INPUT: Input approximate FWHM in pixels
daofind = DAOStarFinder(threshold = threshold, fwhm = FWHM) # make a star finder
```

This cell begins with the boolean flags that determine the rigor of image corrections. Setting all three to *False* performs basic data reduction. The second segment defines two user-input values used to determine bad pixels. The next segment defines a bad pixel correction function that checks for the four pixels around each bad pixel and converts the value of the bad pixel to the average of all available neighbor pixels. The last two segments outline constants used in cosmic ray correction and star identification which can be adjusted by the user.

Block 4

```
# initialize lists of exposure times and filters
object_list = []
object_dict = {}
object_dict_cosmic = {}
exposure_list = []
filter_list = []

img = ccdp.ImageFileCollection(imgpath) # set up a file collection at the image path

# fill the lists of exposure times and filters
for ccd, file_name in img.ccds(imagetyp = 'DARK', ccd_kwargs = {'unit':'adu'}, return_fname = True):
    with open(imgpath / file_name, 'rb') as file:
        exposure_header = ccd.header['EXPTIME']
        if exposure_header not in exposure_list:
            exposure_list.append(exposure_header)
for ccd, file_name in img.ccds(imagetyp = 'FLAT', ccd_kwargs = {'unit':'adu'}, return_fname = True):
    with open(imgpath / file_name, 'rb') as file:
        ccd = CCDData.read(file, unit='adu')
        filter_header = ccd.header['FILTER']
        if filter_header not in filter_list:
            filter_list.append(filter_header)

# iterate through files to identify objects
for ccd, file_name in img.ccds(imagetyp = 'LIGHT', ccd_kwargs = {'unit':'adu'}, return_fname = True):
    with open(imgpath / file_name, 'rb') as file:
        ccd = CCDData.read(file, unit='adu')
        if file_name[:3] != 'TIC':
            break
        obj_name = file_name[:12]
        if obj_name not in object_list:
             object_list.append(obj_name)

# create destination folders for reduced data by object
for object in object_list:
    path = Path(filespath + object[:6])
    path_cosmic = Path(cosmic + '/'+ object[:6])
    if os.path.exists(path):
        shutil.rmtree(path)
    if os.path.exists(path_cosmic):
        shutil.rmtree(path_cosmic)
    os.makedirs(path)
    os.makedirs(path_cosmic)
    object_dict[object] = path
    object_dict_cosmic[object] = path_cosmic

print("Objects: ",object_list)
print("Exposure times: ",exposure_list)
print("Filters: ",filter_list)

# NOTE: RE-RUNNING THIS CELL WILL DELETE FOLDERS OF NEWLY REDUCED DATA AND WILL FORCE YOU TO RERUN THE DATA REDUCTION AND/OR COSMIC RAY CORRECTION
```

This cell makes a number of lists to keep track of unique exposure times, filters, and objects. Each of the first three *for* loops adds units to each type of .fits file, reads keywords from the headers, and adds them to a list if not already found. The third *for* loop for object images reads the object identifier from the file name and overwrites the original unitless files with these new ones. Any files that don't follow the 'TIC*.fits' pattern aren't included with the object lists and will be skipped. The final *for* loop creates folders for each unique object and deletes pre-existing folders of the same name before creating new ones. At the end of this block, AstroPypelyne outputs lists of objects, exposure times, and filters for the user to see.

Block 5

```
# add units to the bias frames
bs = ccdp.ImageFileCollection(imgpath)
for ccd, file_name in bs.ccds(imagetyp = 'BIAS', ccd_kwargs = {'unit':'adu'}, return_fname = True):
    with open(imgpath / file_name, 'rb') as file:
        ccd = CCDData.read(file, unit='adu')
        ccd.write(new_bias_path / file_name, overwrite = True)

# combine bias frames
bsU = ccdp.ImageFileCollection(new_bias_path)
images_to_combine = []
for ccd, file_name in bsU.ccds(imagetyp = 'BIAS', ccd_kwargs = {'unit':'adu'}, return_fname = True):
    with open(new_bias_path / file_name, 'rb') as file:
        ccd = CCDData.read(file, unit='adu')
        images_to_combine.append(ccd)
bscombiner = ccdp.Combiner(images_to_combine)
superBias = bscombiner.average_combine()
superBias.write(new_bias_path / 'superBias.fits', overwrite = True)
```

The first segment of code adds units to the bias frames and writes them to a new folder generated earlier. The other segment combines the bias frames by sending an ImageFileCollection (a form of list in the *ccdproc* library) through an average-based combiner, saving the super bias with the updated bias frames.

Block 6

```
# subtract super bias from darks
dk = ccdp.ImageFileCollection(imgpath)
for ccd, file_name in dk.ccds(imagetyp = 'DARK', ccd_kwargs = {'unit':'adu'}, return_fname = True):
    with open(imgpath / file_name, 'rb') as file:
        ccd = CCDData.read(file, unit='adu')
        with open(new_bias_path / 'superBias.fits','rb') as biasfile:
            superBias = CCDData.read(biasfile, unit='adu')
            dark = ccdp.subtract_bias(ccd, superBias)
            dark.write(new_dark_path / file_name, overwrite = True)
```

Dark frames are collected into an ImageFileCollection, given units, and saved in their new folder. The super bias is also subtracted from each dark frame.

Block 7

```
# initialize list of super darks
superDarks = []

# combine darks based on exposure time
dkU = ccdp.ImageFileCollection(new_dark_path)
for exp in exposure_list:
    images_to_combine = [] # intialize a list of images for use
    for ccd, file_name in dkU.ccds(imagetyp = 'DARK', ccd_kwargs = {'unit':'adu'}, return_fname = True):
        with open(new_dark_path / file_name, 'rb') as file:
            ccd = CCDData.read(file, unit='adu')
            if ccd.header['EXPTIME'] == exp:
                images_to_combine.append(ccd)
    dkcombiner = ccdp.Combiner(images_to_combine)
    superDark = dkcombiner.average_combine()
    new_name = 'superDark' + str(exp) + '.fits'
    superDark.header['EXPTIME'] = exp
    superDark.write(new_dark_path / new_name, overwrite = True)
    superDarks.append(superDark)

if bad_pixel_mask == True:
    super_found_short = False # initialize flags as false
    super_found_long = False
    for superd in superDarks:
        if superd.header['EXPTIME'] == short_dark_xp:
            shortDark = superd
            super_found_short = True
        if superd.header['EXPTIME'] == long_dark_xp:
            longDark = superd
            super_found_long = True
    if super_found_short != True or super_found_long != True:
        print("ERROR: Super Darks could not be found that matched input exposure times. Bad pixel masking will not be performed.")
        bad_pixel_mask = False # prevent code from "correcting" the images with a faulty mask
    if super_found_short == True and super_found_long == True:
        dark_short = shortDark.multiply(gain * u.electron / u.adu).divide(short_dark_xp * u.second)
        dark_long = longDark.multiply(gain * u.electron / u.adu).divide(long_dark_xp * u.second)
        ratio = dark_short.divide(dark_long)
        maskr = ccdp.ccdmask(ratio)
        print("Number of bad pixels identified: ", len(np.where(maskr)[0]))

# check the supers to make sure they're all good
for super in superDarks:
    print(super.header)

    # (optional) bad pixel masking
    if bad_pixel_mask == True:
        fixed_superDark = fix_pix(image=super, mask=maskr)
        super = CCDData(fixed_superDark.data, wcs = super.wcs, unit = u.adu, header = super.header)
```

In the first segment, dark frames are average-combined, forming a super dark per unique exposure time. Each exposure time is used to iterate through a *for* loop where darks of that exposure time is collected into a list for combination. These super darks are saved with the updated dark frames with their exposure time added to their file name. The super darks' headers are also updated to match the dark frames that formed them. If bad pixel masking and correcting is turned on, a dark vs time ratio is created in the middle segment of code. Bad pixels are identified with their locations stored in a mask.

Block 8

```
# tolerance of darks (in seconds) to fit flats
tolerance = 1 # USER INPUT: Adjust this to change how strictly darks are fit to flats

# subtract super bias and super dark from flats
fl = ccdp.ImageFileCollection(imgpath)
for ccd, file_name in fl.ccds(imagetyp = 'FLAT', ccd_kwargs = {'unit':'adu'}, return_fname = True):
    with open(imgpath / file_name, 'rb') as file:
        ccd = CCDData.read(file, unit='adu')
        super_found = False # initialize this flag for checking if a correct super has been found
        flat = ccdp.subtract_bias(ccd, superBias)
        for super in superDarks:
            if super.header['EXPTIME'] - ccd.header['EXPTIME'] < tolerance and super.header['EXPTIME'] - ccd.header['EXPTIME'] > (tolerance * -1):
                superDark = super
                super_found = True
        if super_found != True:
            print("ERROR: No super dark found within tolerance for ", file_name, " with exposure time ", ccd.header['EXPTIME'])
            print("Please adjust the tolerance or ensure the correct Dark.fits files are in the directory.")
            break
        flatter = ccdp.subtract_dark(flat, superDark, exposure_time = 'EXPTIME', exposure_unit = u.second)
        flatter.write(new_flat_path / file_name, overwrite = True)
```

In this segment, flats have the super bias subtracted from each of them. A super dark is then identified for the flat within the specified tolerance. The identified super dark then has the dark that best fits it subtracted from it. If no super darks are identified within the tolerance defined, an error message is output and the exposure time printed for the user.

Block 9

```
# initialize list of super flats
superFlats = []

# combine flats based on filter
flU = ccdp.ImageFileCollection(new_flat_path)
for filter in filter_list:
    images_to_combine = [] # initialize a list of images for use
    for ccd, file_name in flU.ccds(imagetyp = 'FLAT', ccd_kwargs = {'unit':'adu'}, return_fname = True):
        with open(new_flat_path / file_name, 'rb') as file:
            ccd = CCDData.read(file, unit='adu')
            if ccd.header['FILTER'] == filter:
                images_to_combine.append(ccd)
    flcombiner = ccdp.Combiner(images_to_combine)
    superFlat = flcombiner.average_combine()
    mean = np.average(superFlat)
    superFlatter = superFlat / mean
    superFlat = CCDData(superFlatter.data, mask = superFlatter.mask, unit = u.adu)
    new_name = 'superFlat' + filter + '.fits'
    superFlat.header['FILTER'] = filter
    superFlat.write(new_flat_path / new_name, overwrite = True)
    superFlats.append(superFlat)

# check the supers to make sure they're all good
for super in superFlats:
    print(super.header)

    # (optional) bad pixel masking
    if bad_pixel_mask == True:
        fixed_superFlat = fix_pix(image=super, mask=maskr)
        super = CCDData(fixed_superFlat.data, wcs = super.wcs, unit = u.adu, header = super.header)
```

Flats are combined into separate super flats, distinguished by their different filters. In order to do this, each filter is used to iterate through a *for* loop, collecting the flats with matching filters. Each collection of flats is average-combined into a super flat before it's normalized, having every value

divided by the mean. Each super flat is saved with the other updated flats with its filter added to its file name. Bad pixel corrections are performed on the super flats if that is turned on.

Block 10

```
# tolerance of darks (in seconds) to fit super darks
tolerance = 1 # USER INPUT: Adjust this to change how strictly darks are fit to images

# (optional) bad pixel masking - superBias
if bad_pixel_mask == True:
    with open(new_bias_path / "superBias.fits", 'rb') as file:
        superBias = CCDData.read(file, unit='adu')
        fixed_superBias = fix_pix(image=superBias, mask=maskr)
        superBias = CCDData(fixed_superBias.data, wcs = superBias.wcs, unit = u.adu, header = superBias.header)

# reduce the data
for ccd, file_name in img.ccds(imagetyp = 'LIGHT', ccd_kwargs = {'unit':'adu'}, return_fname = True):
    with open(imgpath / file_name, 'rb') as file:
        ccd = CCDData.read(file, unit='adu')
        if file_name[:3] != 'TIC':
            break
        wcs_info = ccd.wcs # save WCS info for the image slicing
        header_info = ccd.header # save header info for later

        # (optional) bad pixel masking
        if bad_pixel_mask == True:
            fixed_pix = fix_pix(image=ccd, mask=maskr)
            ccd = CCDData(fixed_pix.data, wcs = wcs_info, unit = u.adu, header = header_info)

        # bias subtraction
        raw_stars_minus_bias = ccdp.subtract_bias(ccd, superBias)

        # dark subtraction
        super_found = False # initialize this flag for checking if a correct super has been found
        for superd in superDarks:
            if superd.header['EXPTIME'] - ccd.header['EXPTIME'] < tolerance and superd.header['EXPTIME'] - ccd.header['EXPTIME'] > (tolerance * -1):
                superDark = superd
                super_found = True
        if super_found != True:
            print("ERROR: No super dark found within tolerance for ", file_name, " with exposure time ", ccd.header['EXPTIME'])
            print("Please adjust the tolerance or ensure the correct Dark.fits files are in the directory and try again.")
            break # will abort data reduction for the named file
        raw_stars_minus_bias_minus_dark = ccdp.subtract_dark(raw_stars_minus_bias, superDark, exposure_time = "EXPTIME", exposure_unit = u.second)

        # flat correction
        super_found = False # initialize this flag for checking if a correct super has been found
        for superf in superFlats:
            if superf.header['FILTER'] == ccd.header['FILTER']:
                superFlat = superf
                super_found = True
        if super_found != True:
            print("ERROR: No super flat found for ", file_name, " with filter ", ccd.header['FILTER'])
            print("Please ensure the correct Flat.fits files are in the directory and try again.")
            break # will abort data reduction for the named file
        corrected_stars = ccdp.flat_correct(raw_stars_minus_bias_minus_dark, superFlat)

        # Save the corrected data in a folder for the right object
        new_object_path = object_dict[file_name[:12]]
        corrected_stars = CCDData(corrected_stars.data, wcs = wcs_info, unit = u.adu, header = header_info) # comment this line out if data cubes are des
        corrected_stars.write(new_object_path / file_name, overwrite = True)
        corrected_stars.write(cosmic_path / file_name, overwrite = True) # save a copy of the stars here for optional cosmic corrections
```

Each object image is taken through the whole reduction process with super darks and super flats matched to the image by exposure time and filter respectively. If bad pixel corrections are turned on, they're performed before any reduction steps are performed. After all reduction steps are performed, the object identifier is read from the file name and the reduced data is saved in the folder

corresponding with that object identifier as referenced in the dictionary containing both full object identifier and the abbreviated form.

Block 11

```
# (optional) Find star positions for masking
if psf_mask == True:
    cosmimg = ccdp.ImageFileCollection(cosmic_path)
    for ccd, file_name in cosmimg.ccds(ccd_kwargs = {'unit':'adu'}, return_fname = True):
        with open(cosmic_path / file_name, 'rb') as file:
            ccd = CCDData.read(file, unit='adu')
            if file_name[:3] != 'TIC':
                break
            stars = daofind(ccd.data) # identify star locations
            positions = np.transpose((stars['xcentroid'], stars['ycentroid']))
            wcs = WCS(ccd.header)
            for i in range(int(positions.size/2)-1):
                ra,dec = wcs.all_pix2world(positions[i,0],positions[i,1],1)
```

If point spread function (PSF) masking is turned on, this will identify stars in each image as point spread functions, taking note of their positions in a new list. In the last *for* loop, pixels are converted to celestial coordinates right ascension (RA) and declination (Dec).

Block 12

```
# (optional) Cosmic Ray Corrections
if cosmic_correct == True or psf_mask == True: # makes this cell run super fast if no cosmic correction is selected
    aperture_size = 10 # USER INPUT: Adjust the number of pixels (above and below centroid value) to be masked in PSF masking
    cosmimg = ccdp.ImageFileCollection(cosmic_path)
    for ccd, file_name in cosmimg.ccds(ccd_kwargs = {'unit':'adu'}, return_fname = True):
        with open(cosmic_path / file_name, 'rb') as file:
            ccd = CCDData.read(file, unit='adu')
            if file_name[:3] != 'TIC':
                break
            if cosmic_correct == True and psf_mask == False: # don't run corrections here if masking for stars is active
                # cosmic ray correction
                ccd_prep = ccdp.gain_correct(ccd, gain * u.electron / u.adu)
                cosmic_ccd = ccdp.cosmicray_lacosmic(ccd_prep, readnoise = read_noise, sigclip = 5, objlim = 30) # identify the cosmic rays
                cosmic_corrected_stars = fix_pix(image=ccd, mask=cosmic_ccd.mask)

            if psf_mask == True:
                # create mask at locations of stars
                image_shape = ccd.shape
                width, height = image_shape
                xmodcent = height - stars['ycentroid']
                ymodcent = width - stars['xcentroid']
                xmodcent = stars['xcentroid']
                ymodcent = stars['ycentroid']
                star_mask = np.ones(image_shape, dtype = 'bool')
                for i in range(stars['xcentroid'].size):
                    star_mask[int(xmodcent[i])-aperture_size:int(xmodcent[i])+aperture_size,int(ymodcent[i])-aperture_size:int(ymodcent[i])+aperture_size
                #ccd_prep = ccdp.gain_correct(ccd, gain * u.electron / u.adu)
                #masked_data = np.ma.masked_array(ccd_prep.data, star_mask)
                #masked_image = CCDData(masked_data, wcs = wcs_info, unit = u.electron)
                #cosmic_ccd, cosmic_mask = ccdp.cosmicray_lacosmic(masked_data.data, readnoise = read_noise, sigclip = 5, objlim = 30) # identify the cos
                # apply cosmic_mask to data
                 # apply cosmic_mask to data
                 masked_corrected_image = ccd.data * star_mask
                 cosmic_corrected_stars = CCDData(masked_corrected_image, unit=u.adu)

            # Save the corrected data in a folder for the right object
            new_object_path = object_dict_cosmic[file_name[:12]]
            cosmic_corrected_stars = CCDData(cosmic_corrected_stars.data, wcs = wcs_info, unit = u.adu) # comment this line out if data cubes are desired
            cosmic_corrected_stars.write(new_object_path / file_name, overwrite = True)
```

This optional block performs all cosmic ray corrections to the extent that the user has defined. If

cosmic ray corrections are activated without PSF masking, then the data is given the correct units for the function and corrections are performed using LaCosmic. If PSF masking is turned on, then prior to cosmic ray corrections, stars are masked out while the cosmic ray mask is created. After the cosmic ray mask is created, it's applied to the data. At the conclusion of corrections, the data is saved to a new folder separate from the data processed with basic reductions.

NOTE:

Cosmic ray corrections isn't fully polished, but LaCosmic works when parameters are adjusted properly. Bad pixel corrections appear to work, but the function doesn't account for adjacent bad pixels. PSF masking is not yet functional in this version of AstroPypelyne, but the groundwork is laid out.

Bibliography

Astropy Collaboration et al. 2013, A&A, 558, A33

—. 2018, AJ, 156, 123

—. 2022, ApJ, 935, 167

Bradley, L., et al. 2023, astropy/photutils: 1.8.0

Craig, M., et al. 2017, astropy/ccdproc: v1.3.0.post1

Harris, C. R., et al. 2020, Nature, 585, 357

McCully, C., et al. 2018, astropy/astroscrappy: v1.0.5 Zenodo Release

OpenAI. 2023, ChatGPT

Tody, D. 1986, in Instrumentation in Astronomy VI, ed. D. L. Crawford (SPIE)

van Dokkum, P. 2001, Publications of the Astronomical Society of the Pacific, 113, 1420–1427

Index