Survey and Data Reduction of Blazars Using the ROVOR Telescope

Nicholas Van Alfen

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

Bachelor of Science

Joseph Moody, Advisor

Department of Physics and Astronomy

Brigham Young University

April 2019

ABSTRACT

Survey and Data Reduction of Blazars Using the ROVOR Telescope

Nicholas Van Alfen
Department of Physics and Astronomy, BYU
Bachelor of Science

Blazars are active galactic nuclei whose jets point directly at the Earth. By observing the variability in these objects, we can determine their morphology and better understand the distribution of material around a blazar. However, only a handful of blazars are studied regularly. To identify potential candidates for a more in depth study in the future, we observed 192 of the brightest blazars in the northern hemisphere for a year in optical wavelengths with our 16" ROVOR telescope. From our observations, we found 13 blazars displaying a significant level of variability and identified a clear bimodal distribution between smooth an stochastic variability.

Keywords: Blazar, Quasar, Active Galactic Nuclei

ACKNOWLEDGMENTS

I would like to acknowledge my advisor, Dr. Joseph Moody, for his help and guidance in all of my undergraduate schooling. Dr. Moody helpedme with more than just my research, but he gave me advice on any area of school and life I asked him about.

I would also like to acknowledge my high school physics teacher Mr. Newton for his enthusiasm and role in getting me interested in physics.

Contents

List of Figures

Chapter 1

Introduction

1.1 Blazars

An Active Galactic Nucleus (AGN) is the area at the center of a galaxy containing a supermassive black hole and light-emitting jet. The structure of an AGN, as seen in Fig. 1.1, includes a torus of hot, ionized material in an accretion disk orbiting the black hole. This hot, spinning, ionized material creates magnetic field lines along a jet that ejects any charged particles that may break off from the accretion disk, producing light. Depending on the type of galaxy and the viewing angle, astronomers have given different names to AGN. For example, a blazar is an AGN whose jet is pointed directly at the Earth. From this jet, the brightness of a given AGN is determined; Telescope measurements yield brightness measured in magnitudes, a logarithmic measurement of how bright it appears to us on Earth.

Recently, the existence of an accretion disk has been imaged directly by NASA in the first ever picture of a black hole taken on April 10, 2019. This image is shown in Figure 1.2.

Like most objects in the sky, an AGN varies in brightness over time. While discussing AGN, this brightness variation is called flaring. Some possible causes of flaring are known, such as

material from the accretion disk falling into the magnetic field lines, an instability in the AGN causing the jet to "wobble." In addition, it has been suggested that relativistic Doppler boosting could account for some of the smooth variability (D'Orazio et al. 2015). Among these several possible mechanisms for flaring, the specific mechanism at work for a single AGN is not necessarily known. This flaring also differs for each blazar in terms of how much it flares, how long it takes to flare, and how regularly this flaring occurs. Some AGN vary smoothly and others stochastically, suggesting either two distinct variability classes or a spectrum of variability between the two extremes. A clear bimodality, as suspected, would confirm distinctly different flaring mechanisms are at work as opposed to a uniform mechanism subject to a spectrum of flaring patterns.

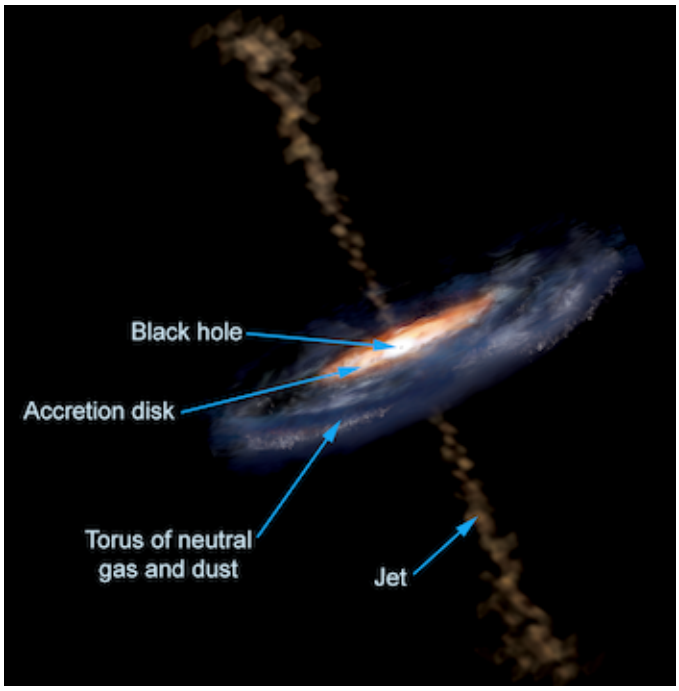

Figure 1.1 The structure of an AGN. A torus of ionized material in an accretion disk orbits the black hole. Relativistic jets account for the observed light. The exact distribution of dust around an AGN is not known, although the general distribution around the accretion disk is accepted. Image courtesy of NASA[1].

[1] https://imagine.gsfc.nasa.gov/science/objects/active_galaxies1.html

[2] https://www.nasa.gov/mission_pages/chandra/news/black-hole-image-makes-history/

Figure 1.2 The first ever direct image of a black hole taken of M87. The accretion disk of this AGN is clearly visible. Image taken April, 10, 2019. Image courtesy of NASA[2].

1.2 Motivation

Over the years, relatively few blazars have been studied regularly or in depth, leaving the astronomical community data starved in regards to most of the observable AGN. While the community has a good general idea of AGN, many specifics remain to be explained. To better understand the physics and matter distribution around black holes, an expanded survey is needed, beyond those well studied AGN, which are referred to as classics or classic blazars. This broader range of study may allow identification of AGN behavior not previously observed in classic blazars and help identify a smaller subset that show interesting behavior. Such a study can then lead to a better model for the structure of an AGN.

1.3 Previous Work

Many previous blazar surveys have used a range of high energy photometric filters. A photometric filter is a filter placed in front of the telescope camera, called a charged-coupled device (CCD), that only allows a certain range of wavelengths of light to pass through, limiting astronomical observations to only one range of wavelengths, known as a color band. This limiting of color bands allows us to study the different regions of the electromagnetic spectrum and compare their behaviors in different bands. Past observation have generally focused on high-energy observations, using filters to allow in x-ray and other high-energy wavelengths of light. Thus, the flaring patterns have only been measured in high-energy regions of the spectrum and not much in the visible spectrum.

Previous work has been unable to identify the exact cause of flaring, though several papers have proposed possibilities. Raiteri et al. (2017) describe an inhomogeneous jet as a cause for observed flaring (Raiteri et al. 2017). Raiteri's inhomogeneous jet model is illustrated in Figure 1.3 which is Figure 4 from Raiteri et al. (2017). This model shows how different regions of the jet are responsible for emitting certain wavelengths. Depending on orientation and alignment with the jet, stronger faring in certain wavelengths may be observed. In Figure 1.3, the two different eyes represent two different alignments, and each observes a different wavelength to have more prominent activity. The instability in the jet causes this orientation to periodically shift, resulting in the time variability we observe.

While an inhomogeneous jet model may describe some of the activity, especially periodic behavior, other activity may result from the distribution of material around the black hole. Figure 1.1 shows the general structure of an AGN, but the exact distribution of material is still unknown. Irregular infall of material from the accretion disk likely causes the more stochastic flaring, and may be affected by the distribution of matter around the black hole and accretion disk.

Plenty of questions about AGN remain: How quickly are they born? When did they come into

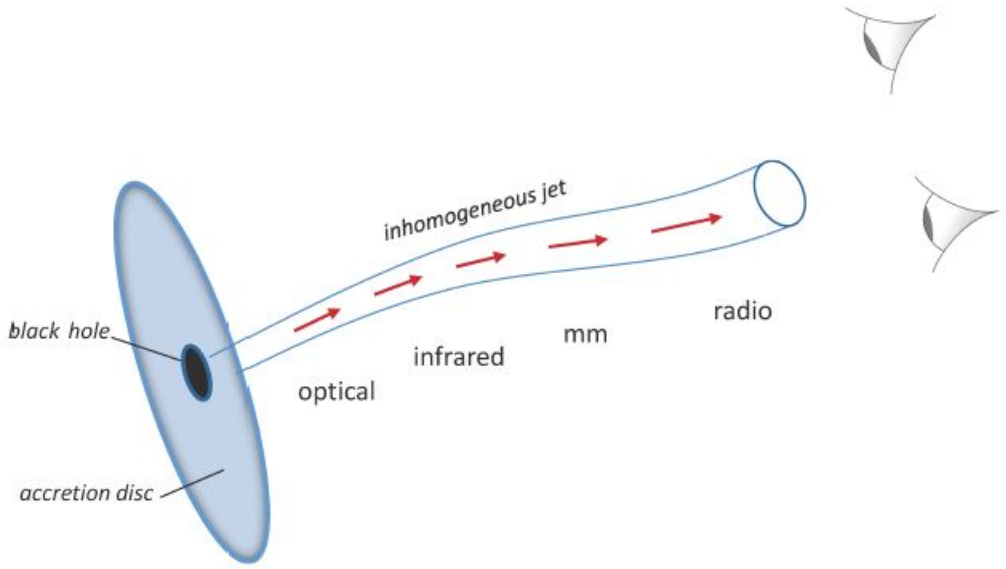

Figure 1.3 Inhomogeneous jet model from Figure 4 of Raiteri et al. (2017). This model proposes that different frequencies of light come from different regions of the jet. Each region has a different orientation relative to the observer as a result of the instability of the jet causing it to curve. The two eyes shown in the figure represent to different alignments with the jet, and each will see a different behavior based on the orientation from which they view the jet.

being? What is the source of these supermassive black holes? etc. Currently, the only information known about AGN comes from the flaring seen here on Earth. By better understanding the physics behind the different types of flaring, we can better understand the structure of the different AGN. By understanding the structure, we can learn about the matter distribution and start to answer some of those fundamental questions. As stated earlier, we plan to approach this problem by gathering a wide range of data in order to capture behavior missed by the more in depth, smaller-focus surveys.

1.4 Previous Work at BYU

The immediate precursor to this work was a thesis written by Lauren Hindman studying the faring rates of these same blazars (Hindman 2018). Lauren's work came from the same study and some of the figures included in her thesis were produced by the code I developed for this study. In her thesis, Lauren found a flaring rate of once every fifteen years on average. This she found by taking the number of blazars found to flare in this survey over our one years of observation and comparing it to the number of blazars surveyed over our one year observation. While this frequency neglects the difference between smooth and stochastic variability, this survey follows up and explores these two flaring modes.

1.5 Overview

The current knowledge about blazars, their flaring patterns, and their matter distribution comes from observations on just a handful of AGN. In order to tackle this problem, we chose as large a list of target AGN as possible. Chapter 2 outlines our surface level survey of a large sample to identify interesting behavior, especially in less frequently studied blazars, to set up the ground work for a later, more in depth observation. Our target list originally consisted of 192 blazars visible in the Northern Hemisphere by our ROVOR telescope. Due to bad weather and insufficient observation points on a few objects, our list of usable blazars contained only 161 blazars.

To define "interesting behavior," we developed a criterion to determine whether a blazar was variable or not during our observation. Based on this criterion, described in Chapter 2, we found thirteen variable blazars (blazars marked as interesting following our criterion). Of those thirteen, we saw two distinct modes of variability, smooth and stochastic, which will be defined later in Chapter 3.

Our results generated a list of blazars for future, more in-depth observation. This will allow

a more focused study of promising targets while including blazars that have not been typically studied as in depth. Future work will use these variable blazars as targets and delve deeper into the causes behind the flaring in addition to the different flaring patterns.

Chapter 2

Data and Methodology

To better understand the structure of a blazar, the causes of flaring must first be studied. Because of the nature of the research, we collected large quantities of data concerning the magnitudes of our objects at different times throughout the year. Reducing and calculating this data required extensive use of programming to speed the calculations and allow the same process to be done for multiple data sets without any mistakes. I used three programming languages for these programs: Mathematica, VBA (Visual Basic for Applications), and Python. I chose Mathematica for it's powerful computational ability and for its fitting features for the all-sky solutions. I chose VBA for many of the later programs because of its integration into Microsoft Excel, allowing new research students to reduce the data in a familiar environment with plots that would update as they did so. Eventually, I converted the VBA code into Python to allow for better, more complex processes. I chose Python for its wide use in science, its compatibility with Linux, its strong computational and graphing abilities, and its open source nature. My Python script, known as the Blazar Reduction and Analysis Code (BARC), replaced all of the calculations done previously by VBA and added new interactive abilities using the ipython terminal.

2.1 Choosing Targets

Before observations began, we had limit our target list to objects visible to our equipment. For our observations, we used BYU's 16" Remote Observatory for Variable Object Research (ROVOR) telescope located near Delta, Utah. Because of ROVOR's relatively small size, it is limited to observing relatively bright objects (see Figure 2.1). ROVOR is a member of the Whole Earth Blazar Telescope (WEBT) network of telescopes, observing blazars, especially those with high energy emissions, around the world, making it a good tool to study blazars (M. Villata 2002).

Figure 2.1 BYU's ROVOR telescope located near Delta, Utah. ROVOR was placed near Delta because of the relatively low levels of light pollution and is controlled remotely. ROVOR is a 16" (0.4 m) telescope, making it a good tool for observing bright objects. ROVOR is a member of the Whole Earth Blazar Telescope (WEBT) contributing to gathering data on blazars.

A year of observations began in the summer of 2015 with targets chosen from the American

Ephemeris, the Veron Cetty-Veron AGN catalogue (Veron-Cetty & Veron 2010), and the WEBT list of high-energy blazars. We chose targets brighter than 16th magnitude (with a magnitude of 16 or lower, as lower number means brighter) as well as a declination of 0 or higher (meaning it is in the Northern Hemisphere)(Van Alfen et al. 2018). These criteria generated a list of 192 blazars, although we found during our observations that many of them were too dim to obtain reliable data.

Once targets were found, we set up a finder chart for observations (see Figure 2.2). A finder chart is an image of the sky containing the target with comparison stars marked. When observing an object in astronomy, comparison stars are needed in order to calibrate results. The same set of comparison stars was observed every time we observed a blazar to differentiate between blazar variability and changing observing conditions. The objects chosen as comparison stars are stable, meaning they do not vary, and should maintain the same magnitude throughout the night. If several of the comparison stars varied, it was likely due to weather or other factors that would affect our blazar equally, allowing us to subtract out the noise.

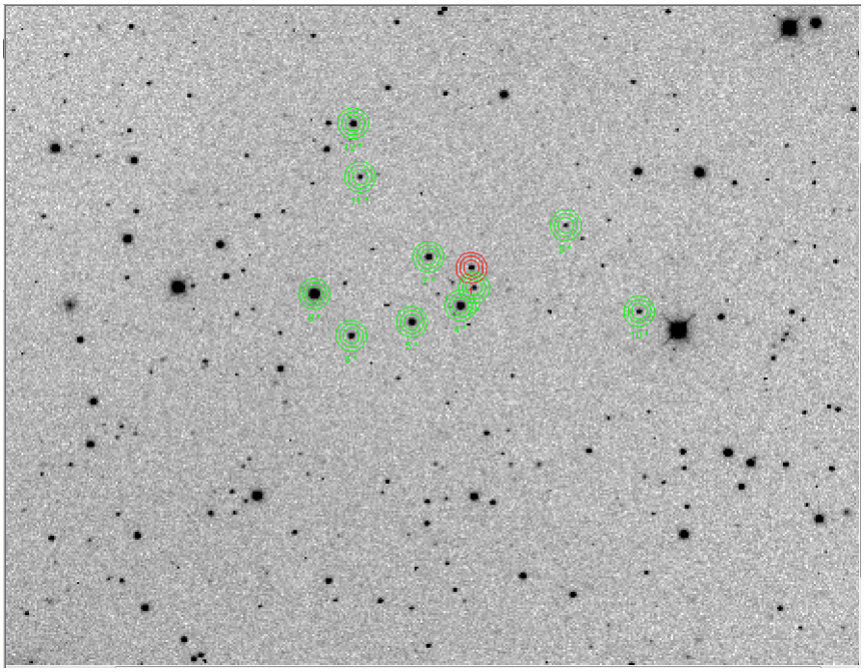

Figure 2.2 An example finder chart. The red annulus marks our blazar while the green annuli mark other bright objects in the field with which to compare. Finder charts ensure that we always observe the same comparison stars each time to stay consistent.

2.2 Calibration Frames

Throughout the year, we observed our target list as often as weather permitted. We observed using Johnson V and R filters, letting in only green and red light, respectively. This process of measuring the brightness in different colors is called photometry. Data reduction was performed to standardize our measurements.

In astronomy, several image types are taken during observations: zeros, darks, flats, and lights. The process of applying these calibration frames to our light frames is known as data reduction. Data reduction eliminates as many artefacts of the instrumentation as possible that may vary from time to time, ensuring as consistent measurements as possible.

As mentioned earlier, a CCD camera takes the frames. A CCD is made of small wells that hold charge. The amount of charge in each well determines how bright that pixel will be and

increases whenever a photon hits it. On an ideal CCD, these wells would only gain charge from the photons emitted by the object and each well would be equally sensitive. Unfortunately, CCD's are imperfect and if not corrected for, these inconsistencies could ruin our data. To correct for this, three types of calibration frames are taken, mentioned above, and applied to each light frame.

When turned on, the wells on a CCD fill to a certain level of charge, called a bias level, that has nothing to do with the brightness of the object being observed. This bias level protects the CCD wells from holding a negative potential and causing errors. Zeros, or bias frames, account for this bias level by taking a picture with zero exposure time: no time for light to enter results in reading only the charge level placed in the wells by default. We take 15 to 30 zero frames and average them, subtracting the result from our final light frame to subtract the bias level.

Even after subtracting the bias level, charge is still introduced that is not related to the observations. When taking an astronomical photo, the camera is exposed for a certain amount of time, collecting light, but while this happens, the camera's electronics are also running. The mechanical vibrations, heat, and electricity can all introduce what we call dark current. The dark current excites the CCD wells and adds value to the pixels even though the object being observed is not producing that light. To account for this, we take a dark frame. A dark frame is a frame where we "expose," or run the electronics to take an image for the same length of time as the observations but with the shutter closed. This process ensures that no light adds to the charge in the CCD and the only pixel counts come from the electronics running (as well as the bias level). We take 15 to 30 dark frames, subtract off the average zero frame, average the dark frames, and subtract that from the light frames as well. All of our targets were observed for 60 seconds, so all of our dark frames were taken for 60 seconds as well.

Once the dark frames have been applied, we take flat fields, or flats. In an ideal CCD, each pixel well would have the same response to light, but this is not the case. Some pixels respond much more or less to light than other pixels, especially near the edge of the CCD where the pixels

tend to be much less sensitive. Flats account for this difference in sensitivity. To take a flat, we point out telescope straight up and just east at an azimuth of 270 degrees and an altitude of 89 degrees, as the sun sets. This part of the sky is uniform in brightness, or flat. Unlike the previous two observations, flat field frames use the same filters as the planned observations. We took 15 flats through Johnson V and R filters each night we observed. We average our flat frames, subtract out the zeros and darks, and normalize the remaining frame. In an ideal CCD, after normalizing, each pixel should have a value of one, but in our case, the less sensitive pixels have a value less than one, and the more sensitive pixels have a value greater than one. The light frame in a specific filter (with zeros and darks subtracted out) is then divided by our normalized flat frame of the same filter. Each pixel of the light frame is divided by the corresponding normalized pixel in the average flat field. The values of the less sensitive pixels are divided by a number smaller than one and will increase, while the reverse happens to the more sensitive pixels.

These three calibrations eliminate false information due to the electronics, making observations taken from the same instrument consistent. However, each telescope still measures slightly different values than every other telescope.

2.3 All-Sky Solutions

Even after calibrating observations using zeros, darks, and flats, each telescope still reads different values than others because of differences in lens size, location, etc. To correct for these differences, an all-sky solution is applied to convert instrumental magnitudes (measurements specific to our telescope) to the standard system (a measurement of true magnitudes) (Cameron Pace 2013). In order to perform all-sky solutions, the observed magnitudes of various Landolt standard stars were calibrated to fit their true value. The same calibration used each night to adjust the values of the Landolt stars was then used on our comparison stars and blazars. Landolt stars are stars with

well-known magnitudes observed by Landolt (2009) that can be used to adjust our measurements. To calibrate from ROVOR's system to the standard system, instrumental magnitudes (the magnitudes specific to our telescope) were plotted against airmass (the amount of atmosphere between our telescope and our object) for each Landolt standard star in each filter. This was performed for each area observed each night it was observed. The instrumental magnitude was calculated from the net counts using the equation:

$$\text{Instrumental Magnitude} = -2.5 * \log_{10} \frac{\text{Net Count}}{\text{Exposure Time}} \tag{2.1}$$

with a 60 second exposure for each object. A linear fit is then calculated and the coefficient for the x term of the fit is obtained. The coefficients are then averaged for each object in their respective filters, yielding a single number for the V filter and a single number for the R filter.

We then calculated the extinction-corrected magnitudes r_0 and v_0, which account for the extinction, or dimming in certain colors due to dust and gas, from the equations:

$$\begin{aligned} r_0 &= r_i + \beta\chi \\ v_0 &= v_i + \alpha\chi \end{aligned} \tag{2.2}$$

These equations are applied to each data point where the lower-case "sub-zero" terms are extinction-corrected instrumental magnitudes, the subscript i indicates instrumental magnitudes, χ is airmass, and α and β are the averages of the linear fit coefficients.

In astronomy, color is defined as the difference in magnitudes through a bluer filter and a redder filter. In our case, we used the $V-R$ color where $V-R$ is referred to as the color index. Combining Eq. 2.3 and Eq. 2.4 with the results of Eq. 2.2, μ_{vr}, μ_v, ζ_{vr}, and ζ_v are calculated where the μ terms are color terms and the ζ values are zero points. These color terms correct for the unequal dimming of the different wavelengths of light. The zero point is an offset to adjust the overall dimming of light due to distance, weather conditions, atmospheric contributions, etc. Using Cameron Pace

(2013), the essential equations used for these calculations are

$$\mathrm{V} = v_0 + \mu_v(\mathrm{V}-\mathrm{R}) + \zeta_v \tag{2.3}$$

$$\mathrm{V}-\mathrm{R} = \mu_{vr}(v-r)_0 + \zeta_{vr} \tag{2.4}$$

where the capital letters are Absolute magnitudes (in this case, Landolt standard magnitudes) and all other variables are as described above. At this point in the calculations, each Landolt standard star has an average $(v-r)_0$ value and a known magnitude from the Landolt tables used (Landolt 2009). My Mathematica script made a plot of the Landolt values versus the average observed values for each standard star and used Mathematica's Fit function to generate a linear fit between them. The fit was of the form $y = mx+b$, where m is μ_{vr} and b is ζ_{vr}, with $(v-r)_0$ as our x. The code found the coefficient and constant term to give us our μ_{vr} and ζ_{vr} values.

Using the μ_{vr} and ζ_{vr} values found, the absolute $V-R$ values can be found for each standard star by evaluating Eq. 2.4. This gives each star's absolute $V-R$ which we then used to solve for μ_v and ζ_v in Eq. 2.3. To solve, Eq. 2.3 was first altered by subtracting v_0 from both sides to get $(\mathrm{V}-v_0) = \mu_V(\mathrm{V}-\mathrm{R}) + \zeta_v$, where v_0 is the average extinction adjusted instrumental magnitude for the standard star. In a similar fashion to how μ_{vr} and ζ_{vr} were obtained, our Mathematica script could now perform a linear fit of $(\mathrm{V}-v_0)$ vs $(\mathrm{V}-\mathrm{R})$. As with above, the μ_v and ζ_v values were pulled from their coefficient and constant places in the fit, yielding the color and zero point terms for the V filter of the Landolt stars.

After finding the color term and zero point by using our Landolt stars, the comparison stars' absolute V magnitude can be obtained from Eq. 2.3. The absolute R magnitude is then $R = V-(V-R)$. My code repeated this process for every standard area observed. Results are shown in Table 2.1. Outliers were eliminated that disagreed greatly with the known Landolt data that were made unreliable by external factors such as frost on the mirror the night of observing. Having standardized the measurements, the absolute magnitudes can be compared to any other object.

Table 2.1 All-Sky Solution Results

Date	μ_{vr}	ζ_{vr}	μ_v	ζ_v	Area
Sep. 19, 2015	0.934	-0.094	-0.177	20.024	SA110
Sep. 19, 2015	1.009	-0.091	-0.099	19.961	SA114
Sep. 20, 2015	0.945	-0.121	-0.173	20.013	SA110
Sep. 20, 2015	1.063	-0.081	-0.121	19.947	SA114
Oct. 12, 2015	0.970	-0.086	-0.115	19.957	SA114
Oct. 12, 2015	0.917	-0.095	-0.143	19.959	SA92
Oct. 13, 2015	0.924	-0.110	-0.145	19.960	SA92
Oct. 13, 2015	1.022	-0.110	-0.129	19.954	SA114
Nov. 7, 2015	0.960	-0.167	-0.140	19.954	SA92
Nov. 7, 2015	0.984	-0.132	-0.100	19.966	SA114
Nov 15. 2015	0.941	-0.153	-0.158	19.946	SA92
Dec. 1, 2016	0.932	-0.303	-0.165	19.668	SA92
Mar. 18, 2016	0.936	-0.179	-0.147	19.800	PG0918
Mar. 18, 2016	0.920	-0.152	-0.071	19.664	SA98
Mar. 19, 2016	0.933	-0.223	-0.164	19.767	PG0918
Mar. 19, 2016	0.871	-0.298	-0.050	19.559	SA98
Apr. 2, 2016	0.943	-0.159	-0.150	19.814	PG0918
Apr. 3, 2016	0.943	-0.132	-0.142	19.975	PG0918
Apr. 6, 2016	0.931	-0.134	-0.176	19.824	PG0918
Apr. 7, 2016	0.911	-0.143	-0.173	19.799	PG0918
Jun. 1, 2016	1.029	-0.187	0.053	20.188	SA107
Jun. 25, 2016	1.027	-0.194	0.053	20.278	SA107

2.4 Blazar Data Calculation

After each night of observing, the images obtained were reduced following the procedure described in Sec. 2.2 using a program called Mira. After reduction, we performed photometry on the image, also using Mira. In this case, photometry refers not only to observing the brightness in different colors but also to using a computer to get the quantitative measurement. Photometry involves marking the blazar with an annulus as well as marking certain comparison stars. An example of this can be seen in Figure 2.2. Mira measured the brightness of the blazar and the comparison stars in the field and saved the results to an Excel file along with information like airmass, exposure time, etc.

The process of analyzing the blazars required lengthy calculations and decisions, creating the need for a program to handle the computation in order to minimize time and maximize accuracy. My program, known as the Blazar Analysis and Reduction Code (BARC), read through each file of blazar data and performed the calculations we specified, marking the blazar if it matched the flaring criterion we had decided meant it was worth taking a closer look at.

The procedure BARC carries out is as follows. See appendix C for details.

- Read in raw data taken from telescope
- Calculate the instrumental magnitudes using Eq. 2.1
- Choose the best half of the comparison stars to use (AllStandardCompare)
- Apply all-sky solutions to all of the comparison stars
- Calibrate blazar to true magnitude using comparison stars
- Determine volatility of blazar based on the flring criterion (Eq. 2.5)
- Fit the standardized blazar magnitudes to a 3^{rd} order polynomial

- Place the blazars that satisfied the flaring criterion in a separate dictionary

The flaring criterion was based on the ratio of the standard deviation in the blazar's magnitude over the standard deviation in the blazar's V-R magnitude (see Eq. 2.5). This signaled that the variation in the V and R magnitudes was due to actual flaring in the blazar and not uncertainty in the observations. The blazar's V and R magnitudes should track together, as both are optical wavelengths and come from the same region of the jet (see Figure 1.3): as the magnitude in one filter increases or decreases by some amount, the magnitude in the other filter should follow suit. This ratio was intended to ignore variation in the blazar's magnitude that resulted from any inherent issues in the photometry from night to night. Issues that made one night appear dimmer or brighter should affect both filters equally. This way, if there is no flaring, any variation due to the instrument or weather should not result in a false positive. We created categories of how variable the blazar appeared based on the ratio of standard deviations shown in Eq. 2.5. We called a ratio under two not variable and two or above variable. While this may have cut some slightly variable blazars from our final list, it also decreased the likelihood of a non-variable blazar satisfying our flaring criterion. Those blazars that fell into the variable category were marked for more in depth study.

$$\frac{\sigma_V}{\sigma_{V-R}} \geq 2 \quad \text{and} \quad \frac{\sigma_R}{\sigma_{V-R}} \geq 2 \tag{2.5}$$

I created a blazar class in Python to store each individual blazar and to contain all of the attributes and calculations we might perform on them. The data given to BARC were the output from Mira in either .xlsx or .xls format. Before serious calculations were made, some checks needed to be performed to ensure that the data were sound. The first check made ensured that the number of objects was consistent. Occasionally, Mira would lose one of the marked objects or one of the students performing the photometry would accidentally add or delete an aperture on a frame, leaving us with either too many or too few objects. The Python script was capable of inferring the desired number of objects based on what the most frequent object count was each

night and deleting any objects beyond that. If there were too few objects, BARC alerted the user and precautions set in place later would handle the resulting issues in calculation.

Then BARC calculated the instrumental magnitude for each data point from Eq. 2.1 using a 60 second exposure for each point. If there was an issue in the data, such as zero or negative net counts or exposure time, the calculated value would be flagged as invalid.

After these preliminary calculations, the data were stored in a special data structure, a 3-dimensional array, for ease of access later, and a second error check was performed on the net counts.

2.5 Error Handling

Sometimes, the number f photons hitting the CCD, referred to as net counts, were either zero or negative due to poor viewing conditions. We defined an instance of a net count issue to mean that a data point had a net count that was either zero or negative. When there were net count issues with the blazar being observed, the night was either discarded or re-photometered, but net count issues with comparison stars were potentially fixable.

Looking at these issues, we noted two levels of net count issues: minor and major. A minor net count issue consisted of two or fewer points having an invalid count in one or each filter from different comparison stars. In these cases, BARC was designed to go through and replace these invalid net counts with the average net count of that object from other frames on that night in that filter. A major net count issue occurs if the night in question had either

- 2 or more instances of a net count issue on a single object through one filter
- 3 or more instances overall of a net count issue through one filter

In the case of a major net count issue, BARC alerted the user and excluded these invalid results in later calculations.

After all of the error checks had been performed, BARC compared the secondary standard stars (another name for comparison stars) to see if any of the stars we chose as secondary standards varied. If any of our comparison stars showed variation, it means they were not good and should be ignored. To do this comparison, BARC chose the first comparison star as the base comparison star, or in other words, the star against which the other secondary standards would be compared. BARC would then cycle through all of the other stars and compares their net counts using an altered version of equation used to calculate instrumental magnitude:

$$\text{Instrumental Magnitude} = -2.5 * \log_{10} \frac{\text{NC}_{\text{compare}}}{\text{NC}^*}$$

, where $\text{NC}_{\text{compare}}$ is the net count of the base comparison star, and NC^* is the net count of the other comparison star. The actual result of this equation is of little significance, but the standard deviation in the comparison of a single star throughout the nights determines how stable that comparison star is. If the standard deviation in the comparison of one of the stars exceeded 0.05, that star was marked to be discarded. If more than five comparison stars were discarded, BARC recognized the base comparison star as bad and performed the same process, using the next secondary standard as the base comparison star. This process continued until a comparison star that resulted in five or fewer discarded secondary standards was found. If no base comparison star was found that discarded five or fewer, the user was alerted and if there were no remaining secondary standards, the calculation was terminated.

This process eliminated issues due to inherent uncertainty in the chosen secondary comparison stars, leaving us with only reliable comparison stars, which in turn increases confidence in the calibrated magnitude of our blazars.

From the all-sky solutions calculated earlier, BARC adjusted the observed blazar magnitudes to their absolute magnitudes using the adjusted comparison stars. our blazar could not be adjusted directly, because the blazar's magnitude is inherently variable whereas the comparison stars are stable. The comparison stars stood in as an intermediary, adjusting to their true, stable values then

comparing the blazar's magnitude to that value. After adjusting our comparison stars with the all-sky solution, BARC would find the offset between the instrumental magnitudes, v_0 and r_0, and the absolute magnitudes, V and R, for each frame and apply that adjustment to the blazar, calibrating the target object's magnitude to its true value.

With the true magnitudes now known, The variability could now be calculated. BARC simply stored the results of Eq. 2.5 as the variability in V and R, so any blazar with a variability greater than or equal to 2 (as is the criterion for being called variable) was marked as variable. This narrowed down our initial list of 192 blazars to just 13 that satisfied the flaring criterion (EQ. 2.5).

After determining if the blazar was sufficiently variable, BARC calculated the correlation coefficient, r^2, of each blazar's V and R data to a third order polynomial fit. We chose a third order polynomial because it allowed enough degrees of freedom to approximate a sinusoidal pattern over small periods of time, but not so many that it would overfit. Initially, we arbitrarily decided that an r^2 value greater than or equal to 0.8 would be classified as smooth and less as stochastic. This initial classification had little basis, in terms of the strict numbers chosen to separate the variability classes, but simply provided some classification until we could see what kind of pattern emerged.

This variability (found by the flaring criterion) and the level of smoothness (determined by r^2) are both key factors in our final analysis. Both may provide insight into the structure and matter distribution of a blazar.

Chapter 3

Results and Analysis

Thus far, my research has been focused on gathering more data from lesser studied blazars to analyze and look for different flaring patterns. Of the original 192 blazars, 13 satisfied our flaring criterion in Eq. 2.5,marking them for for future study. We hope to either confirm previously seen behavior or identify new patterns. Either way, this wider survey will provide information beyond the plethora of data taken from classic blazars.

3.1 Findings

Of the 192 objects from our original list of blazars, 161 had sufficient data to be analyzed in the manner described in Chapter 2. The results are presented in Table A.1 taken from Van Alfen et al. (2018). Of the blazars marked as variable, the median Min - Max magnitude values were 0.60 and 0.55 for V and R filters, respectively.

We confirmed thirteen blazars as variable (see Figure 3.1). Four were stochastic and nine were smooth. More detailed information concerning the smoothness of the thirteen variable blazars can be seen in Table 3.1 and data on all of our blazars can be seen in Table A.1. It is important to note that any blazars observed 4 time or fewer were automatically classified as non-variable and marked

with ellipses in Table A.1 because any dataset with 4 points or fewer will automatically have an r^2 value of 1, making their results useless.

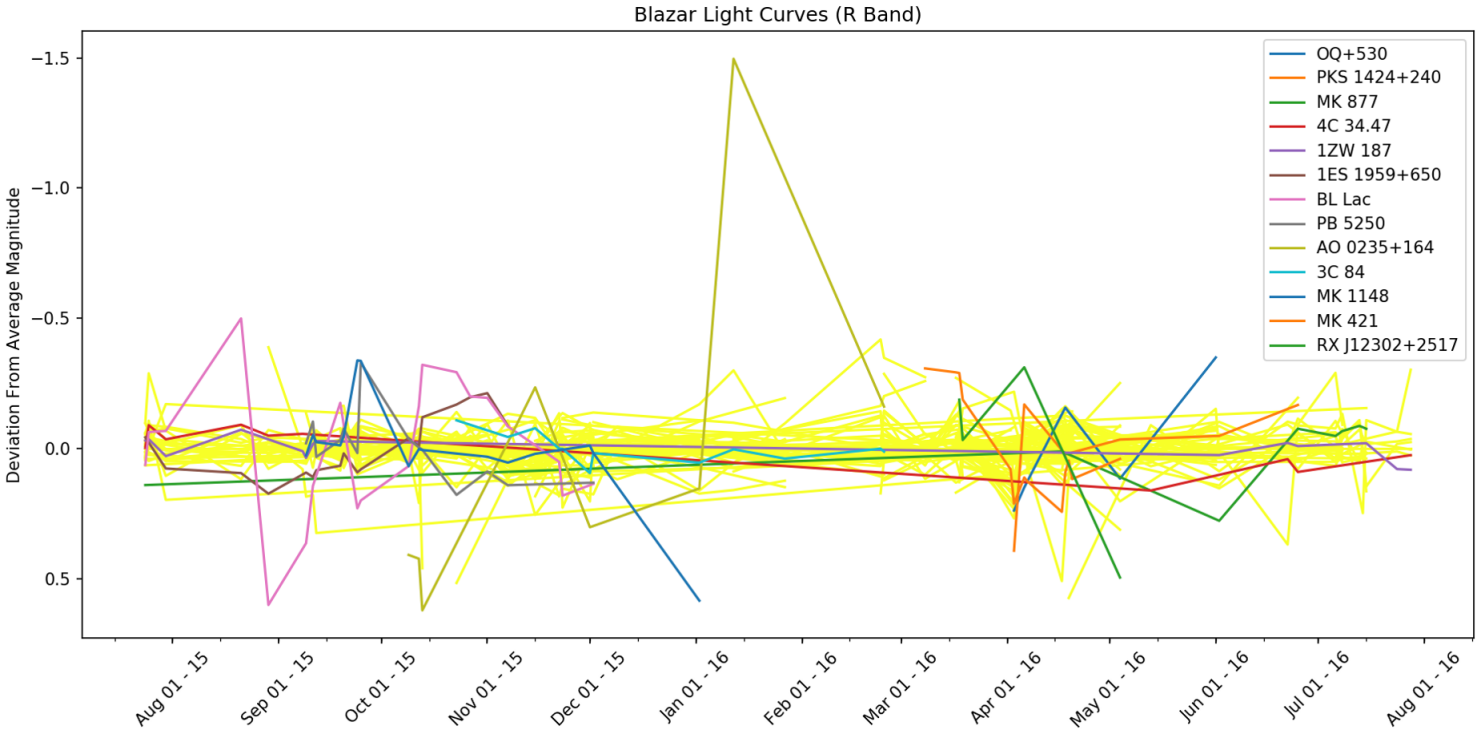

Figure 3.1 Blazars' variance observed in the R band. The y-axis shows how much brighter or dimmer each given point is from its blazar's average magnitude in the R band. The lines in yellow represent the blazars that did not fit the flaring criterion. Those lines with their own unique color are those that the criterion marked as variable and will be followed up on in future research. Note that, although some of the yellow lines deviate significantly,implying potential flaring, they did not satisfy the flaring criterion due to a large standard deviation in the V−R color.

As mentioned above, only 13 varied significantly enough to fulfill the flaring criterion (equation 2.5). The mass of yellow lines in the background of Figure 3.1 are those blazars that did not satisfy the criterion during our period of observation while those with their own unique colored lines are the 13 marked as variable. As previously mentioned, this criterion ensured that the variability we seen due to an actual variability in the blazar's magnitude and not just uncertainty in our observations. The V and R bands flare together, as they both come from the same region in the jet

(see Figure 1.3), so uncertainty in the V−R color warns of error in our instrumental observation, rather than variability in the jet.

Looking at the non-varying blazars in Figure 3.1 (marked in yellow) some show some significant deviation from their mean value. However, these blazars are marked as non-variable because of their large standard deviation in their V−R color. Although they may have had large variability in either V or R, the large V−R standard deviation means that our observations held significant error and thus are unreliable. We don't claim (by Figure 3.1) that the 13 blazars marked as variable are the only blazars varying, but rather that they were the only objects that showed interesting behavior during our observation period according to equation 2.5. By following up with these objects, we hope to see more interesting behavior.

We are especially focused on the object named AO 0235+164 due to its significant flare. As seen in Figure 3.1, AO 0235+164 brightens by 1.5 magnitudes in a span of just 10 days (note that a more negative magnitude is brighter and that magnitude is a logarithmic scale). This 1.5 magnitude brightening far exceeds any other object's flaring during our observation and may be indicative of an irregular infall of material from the accretion disk into the jet's magnetic field lines. While it may be a coincidence that we observed when this happened, if another flare is seen on a similar scale, it could shed light on the structure and physics that would cause such an infall of material to happen multiple times.

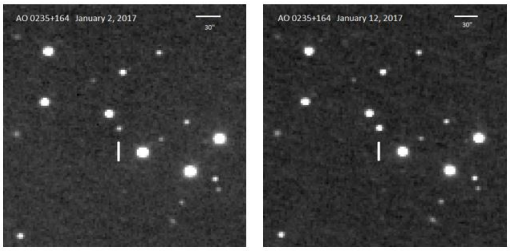

Figure 3.2 The blazar AO 0235+164 shown on January 2, 2017 (left) and ten days later on January 12, 2017 (right). In just 10 days, this blazar brightened by 1.5 magnitudes. Enough to be noticeable by the naked eye in our pictures.

Object AO 0235+164 had a particularly noteworthy flare, brightening by about 1.5 magnitudes between the 2nd and 12th of January 2016, as seen in Figure 3.1. This is shown in Figure 3.2. The brightening is visible even to the naked eye. The blazar in question has brightened noticeably, and to make sure that this was a real result, we went back to the data that night and photed the data again. We verified that the blazar brightened by 1.5 magnitudes while the comparison stars remained unchanged.

The brightening of AO 0235+164 is visible even to the naked eye, as shown in Figure 3.2. To make sure that this brightening was real, we went back to the image obtained that night and photed it again. We verified that the blazar brightened by 1.5 magnitudes while the comparison stars remained unchanged.

As mentioned in Chapter 2, after classifying a blazar as variable, we fit a 3^{rd} order polynomial to the variable blazar's data and found its r^2 value. A smoothly varying blazar would vary sinusoidally, so the r^2 to a 3^{rd} order polynomial would be very high. This is of interest to us because if we see groupings of r^2 values, as we expect to, this would imply distinctly different mechanisms for flaring, suggesting a clear divide between different flaring modes. We arbitrarily set a cutoff of $r^2 = 0.8$ and said anything higher would be classified as smooth and anything lower as stochastic. We did not intend this to be any sort of refined or final cutoff, but rather a preliminary marker for classification.

After measuring these r^2 values, we created a histogram, binned by 0.1 (see Figure 3.3) to see if there was any pattern. A clear bimodality exists to the data, as seen in Figure 3.3. The obvious bimodality indicates that the cutoff of $r^2 \geq 0.8$ should be amended to $r \geq 0.07$, resulting in Table 3.1, but there is such a clear divide that our $r^2 = 0.7$ cutoff plays no real role. In the distribution, there are no blazars with an r^2 value between 0.5 to 0.6 with two groupings on either side, suggesting two entirely separate flaring mechanisms.

More research is needed to discover what exactly these different flaring mechanisms are, but

Table 3.1 Names of the thirteen variable blazars and the r^2 values to a 3^{rd} order polynomial fit in both the V and R filters over one year. Listed in the last column is the blazar's final classification as either smooth or stochastic which can be verified by looking at the r^2 values in V and R. A strong r^2 suggests a strong 3^{rd} order polynomial trend. Originally arbitrarily defined as $r^2 \geq 0.8$ as the cutoff for smooth and anything lower as stochastic, this was later amended to 0.7 after seeing Figure 3.3.

Name	r^2 in V	r^2 in R	Variability
MK 1148	0.33	0.43	Stochastic
AO 0235+164	0.24	0.45	Stochastic
IZw 187	0.41	0.35	Stochastic
BL Lac	0.21	0.22	Stochastic
3C 84	0.82	0.78	Smooth
Mrk 421	0.97	0.96	Smooth
RX J12302+2517	0.97	0.96	Smooth
OQ+530	0.86	0.85	Smooth
PKS 1424+240	0.96	0.97	Smooth
4C 34.47	0.91	0.85	Smooth
MK 877	0.70	0.77	Smooth
IES 1959+650	0.89	0.91	Smooth
PB 5250	0.80	0.74	Smooth

understanding them could give insight into the physics behind blazar variability. Better understanding the causes of flaring can refine the current understanding of the structure around an AGN and gain insight into the conditions of the early universe when these objects formed.

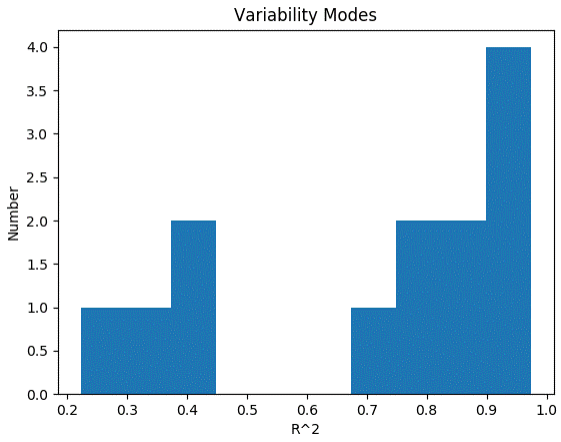

Figure 3.3 A histogram of the r^2 values for R magnitude data fit to a 3rd order polynomial. A clear bimodal distribution is shown in the histogram, with no blazars occupying the space with r^2 of 0.5 or 0.6.

3.2 Conclusion

To better understand the strugture of AGN, we surveyed 192 blazars in search of a smaller subset on which to perform more in depth observations in the future. Of our original 192 blazars, 161 had sufficient data to analyze. Of those 161, only 13 satisfied our flaring criterion: four were classified as stochastic and nine as smooth. Because of the bimodality shown in Figure 3.3, we conclude that there truly exists a bimodal variability distribution rather than a continuous spectrum of variability. Two distinct classes of variability imply two distinct classes of physics behind each type of flaring.

More research needs to be done to determine the exact cause of this bimodal distribution and determine what that means for the distribution of material around an AGN.

3.3 Future Work

Now that a smaller survey of blazars including both some classics and lesser studied blazars has been identified, a more in depth series of observations and analyses can be performed. Future students will be able to focus on these 13 blazars and gather significantly more data points than were gathered for each object in this survey. A similar analysis to this project should be done to verify variability, especially for the less regular, stochastic blazars, before determining the cause of flaring.

Appendix A

Results Table

Table A.1 Photometry of the 161 objects with reliable data. The "..." values in the variability column were given to those blazars that did not satisfy the flaring criterion Equation 2.5.If the blazar was variable during our observation, according to that criterion, the variability column will label it as either smooth or stochastic, depending on the r^2 fit of its observed data to a third order polynomial. Any blazars with 4 or fewer nights of observation were given a variability classification of ... (not variable) due to the fact that 4 or fewer points will always receive an r^2 of 1 for a third order polynomial, making their results meaningless. It is also important to note that the standard deviation in the V and R filters is not a measure of the uncertainty in each magnitude, but rather a measure of how much its magnitude varied.

Name	Avg V	Std Dev	Min - Max	Avg R	std dev	Min - Max	Variability	Nights
4C 25.01	16.05	0.07	0.27	15.75	0.05	0.19	...	13
A 0021+25	15.71	0.05	0.19	15.04	0.04	0.17	...	18
PG 0026+129	15.43	0.07	0.25	15.14	0.04	0.15	...	16
PB 6151	16.19	0.10	0.35	15.82	0.06	0.22	...	14
MK 1148	15.52	0.22	1.04	15.13	0.19	0.92	Stochastic	16
1ZW 1	14.16	0.05	0.24	13.71	0.04	0.16	...	16
PG 0052+251	15.31	0.10	0.45	15.09	0.08	0.31	...	18

Name	Avg V	Std Dev	Min - Max	Avg R	Std Dev	Min - Max	Variability	Nights
PHL 909	16.24	0.08	0.30	15.87	0.03	0.12	...	14
IRAS 01072-0348	15.80	0.07	0.24	15.47	0.06	0.26	...	10
GC 0109+224	15.14	0.15	0.49	14.70	0.14	0.45	...	14
MK 357	15.34	0.07	0.26	15.16	0.04	0.15	...	12
1ES 0120+340	17.47	0.16	0.54	16.61	0.07	0.21	...	10
IRAS 01475+3554	16.50	0.12	0.49	15.96	0.09	0.31	...	14
MK 1014	15.74	0.11	0.44	15.40	0.06	0.23	...	10
MK 586	15.60	0.09	0.37	15.39	0.05	0.14	...	10
3C 59	16.79	0.12	0.49	16.24	0.06	0.21	...	12
PKS 0215+015	18.30	0.32	0.78	19.32	1.46	3.37	...	3
B3 0225+389	18.83	0.54	1.66	17.86	0.41	1.48	...	13
1ES 0229+200	16.88	0.13	0.46	16.14	0.12	0.43	...	12
AO 0235+164	18.41	0.60	2.22	17.50	0.59	2.12	Stochastic	9
S2 0241+62	16.75	0.11	0.44	15.66	0.09	0.31	...	10
4U 0241+61	16.86	0.12	0.53	15.69	0.07	0.24	...	13
3C 84	13.12	0.05	0.17	12.52	0.05	0.20	Smooth	12
3C 110	17.39	0.11	0.22	17.27	0.35	0.70	...	2
MG 0509+0541	15.62	0.28	0.96	15.18	0.25	0.94	...	12
HS 0624+6907	14.43	0.03	0.12	14.09	0.03	0.11	...	9
1ES 0647+250	15.84	0.16	0.48	15.45	0.16	0.44	...	9
MS 07007+6338	15.58	0.03	0.12	15.33	0.03	0.08	...	10
7ZW 118	15.37	0.09	0.35	14.92	0.05	0.16	...	11
B2 0709+370	15.70	0.06	0.16	15.48	0.05	0.14	...	10

Name	Avg V	Std Dev	Min - Max	Avg R	Std Dev	Min - Max	Variability	Nights
4C 41.30	15.68	0.05	0.13	15.54	0.04	0.13	...	10
OI+90.4	17.21	0.18	0.58	16.57	0.10	0.33	...	9
1E0754+39.4	14.65	0.07	0.27	14.37	0.05	0.20	...	10
IRAS 07598+6508	14.67	0.02	0.06	14.45	0.02	0.06	...	9
1ES 0806+524	15.35	0.07	0.19	14.90	0.07	0.20	...	8
PG 0804+761	14.69	0.12	0.46	14.46	0.10	0.38	...	11
US 1329	15.58	0.13	0.49	15.30	0.05	0.18	...	10
CSO 199	16.81	0.09	0.27	16.51	0.08	0.26	...	7
7ZW 244	16.26	0.09	0.33	15.94	0.03	0.10	...	7
SBS 0909+532	16.52	0.07	0.22	15.90	0.04	0.12	...	6
TON 1057	15.39	0.10	0.32	15.06	0.11	0.38	...	7
TON 1078	16.37	0.06	0.17	16.13	0.03	0.09	...	6
4C 12.35	18.69	0.47	1.23	18.53	0.10	0.24	...	4
3C 232	15.82	0.09	0.25	15.52	0.04	0.11	...	5
MK 132	16.23	0.07	0.19	15.93	0.07	0.21	...	5
4C 13.41	15.61	0.03	0.06	15.25	0.02	0.05	...	2
TON 488	17.01	0.12	0.40	16.63	0.15	0.47	...	7
TON 1187	15.98	0.17	0.46	15.55	0.06	0.17	...	5
SBS 1010+535	16.44	0.16	0.50	16.15	0.09	0.31	...	8
TON 34	16.39	0.06	0.16	16.01	0.09	0.24	...	4
B3 1019+397	17.11	0.19	0.67	16.79	0.10	0.31	...	8
MK 142	15.70	0.18	0.50	15.22	0.06	0.16	...	4
SBS 1047+550	16.93	0.18	0.57	16.85	0.10	0.31	...	6

Name	Avg V	Std Dev	Min - Max	Avg R	Std Dev	Min - Max	Variability	Nights
RX J10547+4831	16.12	0.11	0.30	15.79	0.09	0.35	...	8
TON 52	16.63	0.17	0.47	16.39	0.05	0.13	...	4
3C 249.1	15.52	0.04	0.13	15.21	0.03	0.09	...	6
MK 421	13.14	0.22	0.68	12.71	0.19	0.55	Smooth	13
HS 1103+6416	15.87	0.10	0.28	15.43	0.08	0.24	...	5
4C 16.30	16.75	0.02	0.05	17.00	0.32	0.64	...	2
TON 1388	15.01	0.03	0.06	14.76	0.02	0.06	...	3
SBSG1116+518	17.36	0.23	0.66	17.08	0.16	0.52	...	6
TON 580	16.67	0.09	0.25	16.36	0.03	0.08	...	4
MK 180	14.68	0.09	0.26	14.14	0.07	0.23	...	6
RX J11479+2715	16.42	0.08	0.22	16.11	0.09	0.23	...	4
CBS 147	17.93	0.23	0.69	17.51	0.11	0.32	...	6
OM+280	16.70	0.10	0.23	16.18	0.18	0.44	...	3
PG 1151+118	16.30	0.03	0.08	16.01	0.02	0.05	...	3
TON 599	17.03	0.15	0.44	16.63	0.18	0.53	...	6
GQ Com	16.66	0.13	0.29	16.25	0.07	0.17	...	3
PG 1206+459	15.58	0.12	0.30	15.35	0.03	0.09	...	4
PG 1211+143	14.78	0.10	0.30	14.56	0.05	0.15	...	5
1ES 1212+078	16.85	0.12	0.24	16.12	0.08	0.16	...	2
ON+325	14.90	0.05	0.14	14.48	0.06	0.15	...	4
RS 4	16.49	0.00	0.01	16.03	0.05	0.12	...	3
MK 205	15.53	0.38	1.10	14.86	0.04	0.13	...	6
TON 618	15.99	0.02	0.06	15.68	0.04	0.09	...	3

Name	Avg V	Std Dev	Min - Max	Avg R	Std Dev	Min - Max	Variability	Nights
3C 273.0	13.12	0.03	0.06	12.91	0.01	0.02	...	3
RX J12302+2517	16.00	0.23	0.65	15.64	0.28	0.81	Smooth	5
TON 1542	15.07	0.05	0.13	14.60	0.03	0.08	...	4
TON 83	16.77	0.03	0.08	16.48	0.03	0.08	...	4
CSO 151	17.13	0.26	0.78	16.63	0.07	0.22	...	6
SBS 1234+607	18.37	0.43	0.96	17.86	0.09	0.23	...	3
PG 1241+176	16.33	0.02	0.06	15.94	0.01	0.03	...	4
PG 1246+586	16.33	0.09	0.27	15.92	0.07	0.20	...	6
LB 19	15.71	0.04	0.10	15.37	0.03	0.06	...	3
KUV 12491+2932	16.23	0.07	0.16	15.96	0.02	0.04	...	3
Q 1252+0200	16.29	0.11	0.35	16.03	0.09	0.25	...	6
1ES 1255+244	17.29	0.18	0.44	16.73	0.12	0.29	...	3
LB 2522	15.72	0.09	0.21	15.28	0.07	0.19	...	4
PG 1307+086	16.09	0.12	0.38	15.76	0.08	0.26	...	6
TON 1565	15.53	0.05	0.13	15.24	0.04	0.10	...	4
TON 153	15.97	0.10	0.30	15.77	0.08	0.23	...	6
PG 1322+659	15.66	0.03	0.08	15.42	0.02	0.05	...	3
4C 55.27	18.20	0.39	1.04	18.02	0.30	0.78	...	4
TON 730	15.95	0.09	0.29	15.59	0.08	0.25	...	6
MK 662	15.51	0.09	0.28	15.08	0.10	0.32	...	6
PB 4142	16.36	0.13	0.28	15.99	0.09	0.22	...	3
TON 182	16.10	0.13	0.34	15.81	0.12	0.35	...	5
PG 1404+226	15.98	0.04	0.10	15.67	0.05	0.15	...	4

Name	Avg V	Std Dev	Min - Max	Avg R	Std Dev	Min - Max	Variability	Nights
PG 1407+265	15.88	0.10	0.32	15.81	0.09	0.28	...	6
PG 1411+442	14.94	0.08	0.22	14.63	0.07	0.21	...	4
PG 1415+451	15.90	0.08	0.20	15.51	0.06	0.15	...	4
1E 1415+259	17.08	0.18	0.47	16.53	0.04	0.10	...	4
OQ+530	15.67	0.22	0.61	15.13	0.22	0.59	Smooth	5
KUV 14207+2308	16.01	0.06	0.16	15.65	0.06	0.16	...	4
2E 1423+2008	16.84	0.20	0.51	16.39	0.07	0.16	...	4
PKS 1424+240	14.76	0.12	0.37	14.36	0.19	0.56	Smooth	6
MK 813	14.98	0.04	0.12	14.70	0.06	0.14	...	4
TON 202	16.83	0.15	0.38	16.56	0.11	0.34	...	5
MK 1383	14.54	0.05	0.15	14.21	0.06	0.15	...	4
PG 1437+398	16.94	0.07	0.20	16.42	0.05	0.15	...	6
MARK 478	14.71	0.08	0.22	14.36	0.04	0.11	...	4
PG 1444+407	16.07	0.07	0.21	15.75	0.05	0.13	...	5
MK 830	17.31	0.08	0.23	16.78	0.09	0.27	...	6
MK 840	16.51	0.27	0.71	15.90	0.10	0.27	...	5
1H 1515+660	17.09	0.28	0.86	16.82	0.26	0.87	...	7
MCG+11-19-005	15.73	0.03	0.08	15.09	0.03	0.09	...	5
RX J15291+5616	16.59	0.54	1.60	16.35	0.51	1.40	...	6
PG 1538+478	16.05	0.06	0.17	15.82	0.04	0.11	...	5
1ES 1544+820	17.30	0.06	0.18	16.75	0.07	0.19	...	5
SBS 1542+541	17.28	0.03	0.08	17.03	0.05	0.15	...	4
MK 876	14.85	0.05	0.12	14.49	0.03	0.07	...	5

Name	Avg V	Std Dev	Min - Max	Avg R	Std Dev	Min - Max	Variability	Nights
TON 256	15.99	0.05	0.15	15.61	0.08	0.25	...	8
3C 332.0	15.88	0.11	0.36	15.38	0.05	0.15	...	8
MK 877	15.33	0.14	0.49	15.09	0.12	0.36	Smooth	9
KP 77	17.33	0.07	0.19	17.13	0.09	0.27	...	6
HS 1626+6433	16.66	0.06	0.25	16.34	0.07	0.32	...	13
KUV 16313+3931	16.69	0.12	0.47	16.39	0.07	0.30	...	11
RX J17025+3247	16.20	0.24	0.82	15.90	0.15	0.50	...	12
3C 351.0	15.67	0.11	0.34	15.23	0.05	0.14	...	7
RX J17159+3112	15.81	0.06	0.19	15.46	0.04	0.14	...	15
PG 1718+481	15.10	0.05	0.22	14.66	0.02	0.08	...	14
4C 34.47	16.32	0.10	0.31	15.91	0.08	0.25	Smooth	10
H 1722+119	15.52	0.22	0.70	14.95	0.17	0.63	...	15
1ZW 187	15.90	0.06	0.23	15.37	0.04	0.15	Stochastic	15
IRAS 17500+5046	15.24	0.02	0.08	14.83	0.02	0.07	...	14
KAZ 102	16.57	0.07	0.20	16.24	0.05	0.15	...	12
KUV 18217+6419	14.23	0.02	0.08	13.87	0.02	0.07	...	13
PGC 61965	15.09	0.04	0.13	14.69	0.05	0.20	...	13
IRAS 18299+4113	16.22	0.03	0.08	15.73	0.13	0.56	...	14
HS 1946+7658	16.42	0.08	0.37	16.05	0.04	0.17	...	15
1ES 1959+650	14.95	0.13	0.46	14.39	0.11	0.39	Smooth	19
4C 74.26	14.69	0.02	0.06	14.14	0.02	0.07	...	20
MK 509	13.79	0.04	0.15	13.25	0.03	0.08	...	13
PG 2112+059	15.62	0.04	0.11	15.31	0.03	0.10	...	8

Name	Avg V	Std Dev	Min - Max	Avg R	Std Dev	Min - Max	Variability	Nights
2ZW 136	14.77	0.06	0.18	14.41	0.03	0.12	...	12
OX+169	16.18	0.04	0.17	15.82	0.04	0.12	...	21
IRAS 21431-0432	16.54	0.08	0.29	16.05	0.06	0.19	...	16
BL Lac	14.12	0.25	1.16	13.39	0.24	1.10	Stochastic	22
4c 31.63	15.61	0.04	0.13	15.21	0.04	0.18	...	21
ZW II 171 s	15.84	0.08	0.30	15.26	0.05	0.20	...	21
KUV 22497+1439	16.11	0.07	0.21	15.75	0.06	0.25	...	20
4C 11.72	15.86	0.05	0.27	15.41	0.04	0.14	...	19
MK 926	14.66	0.04	0.13	14.07	0.03	0.08	...	10
PB 5235	15.95	0.04	0.13	15.69	0.04	0.16	...	13
PB 5250	15.40	0.15	0.60	14.71	0.12	0.51	Smooth	14
4C 09.72	16.15	0.07	0.26	16.00	0.07	0.24	...	13
3C 465.0	13.67	0.05	0.22	12.96	0.03	0.12	...	15
4C 09.74	16.28	0.11	0.47	16.07	0.05	0.20	...	19
1ES 2344+514	15.38	0.05	0.21	14.62	0.04	0.14	...	18
PKS 2349-014	16.37	0.05	0.16	15.80	0.11	0.48	...	14

Appendix B

All Sky Solutions Code

In addition to surveying the blazars and their comparison stars, we observed several well known stars whose magnitudes are listed in Landolt (2009). By observing these, we can adjust the magnitudes observe by ROVOR and convert them to a true magnitude scale. To do this, we follow the procedures outlined in Section 2.3. I wrote a code in Mathematica to handle the computation to ensure our results were accurate and consistent before applying them to our final calculations. The following pages include this program.

```
(*UPDATE LOG
  5/2/17
     - Fixed createLandolt function so that it threw out individual
  elements of each sub-list instead of some of the sub-lists.

       - Added date features to allow the file writing process to automatically
  add the right date to the folder name. Added functions subtractDay and
  dateConvert as well as variables dateTaken (the MM/DD/YYYY format of the date)
  and timeTag (the converted Year_Month_Day format).

   - Added a choice to delete the old data file after writing to
  a new location instead of automatically deleting in case somehting
  went wrong and you need the file or don't want to delete it yet.

   - Added an override option for safeDirectory create that allows an add-
 on to be passed in as well. Also,
the call to safeDirectoryCreate in the file write portion of the code was
  changed to call to overridden function and all directory create code
  concatenated timeTag onto the origwriteLocation. See function for details.

       - Changed length of Do[] loop to Length[red] in ErrorLog
  writing sction of code. Originally the last object was being
  left our of the writing process. Fix includes the last object.
   ------------------------------------------
  ------------------------------------
  5/3/17
     - HUGE. SWITCHED INDICES USED IN adjustLandolt
  FUNCTION (2 TO 3 IN THE If["R"] CASE AND 3 TO 2 IN ELSE CASE).
*)

location = InputString["Name of File With Star Data: "];
landoltLocation = InputString["Name of File With Landolt Data: "];

SetDirectory[NotebookDirectory[]];
(*data = Import["test.csv"];*)
data = Import[location <> ".csv"];
dataT = Transpose[data];

netcountcol = 17;
exptimecol = 23;
datecol = 19;
dateTaken = data[[2, datecol]];
```

```
(*For when Counts/s and Inst. Mag. haven't already been calculated*)
If[Length[Transpose[data]] == 25,
  countssec = Drop[Transpose[data][[netcountcol]], 1] /
     Drop[Transpose[data][[exptimecol]], 1] // N;
  instmag = -2.5 * Log10[countssec];
  PrependTo[countssec, "Counts/s"];
  PrependTo[instmag, "Inst. Mag."];
  data = Transpose[data];
  AppendTo[data, countssec];
  AppendTo[data, instmag];
  data = Transpose[data];
  dataT = Transpose[data]];

(*Handles boundary cases where the day is the first of the
 month. The files are labled as the day after they were taken,
so the true date is one earlier. Boundary cases handled include first
 of the month as well as first month of the year, cycling to the
 previous month and even the previous year if the month was January*)
subtractDay[day_, month_, year_] :=
 Module[{shiftedMonth = 0, shiftedYear = 0, maxJan = 31, maxFeb = 28,
   maxMar = 31, maxApr = 30, maxMay = 31, maxJun = 30, maxJul = 31,
   maxAug = 31, maxSep = 30, maxOct = 31, maxNov = 30, maxDec = 31, max = 0},
  shiftedMonth = month;
  shiftedYear = year;
  If[shiftedMonth ≠ 1, shiftedMonth--, shiftedMonth = 12;
   shiftedYear--];
  If[shiftedMonth == 1, max = maxJan];
  If[shiftedMonth == 2, max = maxFeb];
  If[shiftedMonth == 3, max = maxMar];
  If[shiftedMonth == 4, max = maxApr];
  If[shiftedMonth == 5, max = maxMay];
  If[shiftedMonth == 6, max = maxJun];
  If[shiftedMonth == 7, max = maxJul];
  If[shiftedMonth == 8, max = maxAug];
  If[shiftedMonth == 9, max = maxSep];
  If[shiftedMonth == 10, max = maxOct];
  If[shiftedMonth == 11, max = maxNov];
  If[shiftedMonth == 12, max = maxDec];
  {shiftedYear, shiftedMonth, max}]

(*Converts from a MM/DD/YYYY format to Year_Month_Day format for file writing*)
dateConvert[date_String] :=
```

```
 Module[{newDates = {}, i = 1, altered = "", year = "", month = "", day = ""},
  While[StringTake[date, {i}] ≠ "/", month = month <> StringTake[date, {i}];
   i++];
  i++;
  While[StringTake[date, {i}] ≠ "/", day = day <> StringTake[date, {i}];
   i++];
  i++;
  Do[year = year <> StringTake[date, {j}], {j, i, StringLength[date]}];

  day = ToExpression[day];
  If[day ≠ 1, day--,
   newDates = subtractDay[day, ToExpression[month], ToExpression[year]];
   year = ToString[newDates[[1]]]; month = ToString[newDates[[2]]];
   day = ToString[newDates[[3]]]];
  day = ToString[day];
  month = ToString[month];
  year = ToString[year];
  If[month == "1", month = "Jan"];
  If[month == "2", month = "Feb"];
  If[month == "3", month = "Mar"];
  If[month == "4", month = "Apr"];
  If[month == "5", month = "May"];
  If[month == "6", month = "Jun"];
  If[month == "7", month = "Jul"];
  If[month == "8", month = "Aug"];
  If[month == "9", month = "Sep"];
  If[month == "10", month = "Oct"];
  If[month == "11", month = "Nov"];
  If[month == "12", month = "Dec"];
  If[StringLength[day] == 1, day = "0" <> day];
  altered = "_" <> year <> "_" <> month <> "_" <> day;
  altered]

(*landoltData = Import["landolt.csv"];*)
landoltData = Import[landoltLocation <> ".csv"];
landoltDataT = Transpose[landoltData];

objcol = dataT[[3]];
objcol = Drop[objcol, 1];
objects = {objcol[[1]]};
titles = data[[1]];

(*Adds new labels to titles for when they are re-
```

```
 merged to output to csv format*)
AppendTo[titles, "r0 or v0"];
AppendTo[titles, "Adjusted R or V"];

Vresults = {};
Rresults = {};
objectindex = 3;
filterindex = 18;
airmassindex = 22;
instmagindex = 27;

(*Checks to see if a list has a value*)
contains[val_, set_List] := Module[{has = False},
  Do[If[val == set[[i]], has = True], {i, 0, Length[set]}];
  has]

(*Creates a list of how many objects there are*)
Module[{},
 Do[If[! contains[objcol[[j]], objects], AppendTo[objects, objcol[[j]]]],
  {j, 0, Length[objcol]}]]

(*Splits data into two lists, one for each filter R and V,
and each of those lists conatins a list for each object,
each of which has a list of the data for that object in that filter
 Object number corresponds to the first index in each of Vdata and Rdata*)
dataSplit[filter_String] := Module[{tempdata = {}, tempindex = 1},
  Do[AppendTo[tempdata, {}], {i, 1, objects[[Length[objects]]]}];
  Do[tempindex = data[[i, objectindex]];
   If[data[[i, filterindex]] == filter,
    AppendTo[tempdata[[tempindex]], data[[i]]]], {i, 2, Length[data]}];
  tempdata]

(*Gets a list of {airmass, intrumental magnitude} pairs for each object
 in a given filter in order to plot and extract coefficient from*)
xyPairUp[selected_List] := Module[{templist = {}},
  If[selected[[1, 1, filterindex]] == "R",
   Do[If[Rdata[[i]] ≠ {}, AppendTo[templist, Transpose[{Transpose[Rdata[[i]]][[
         airmassindex]], Transpose[Rdata[[i]]][[instmagindex]]}]],
     AppendTo[templist, {}]], {i, 1, Length[selected]}],
   Do[If[Vdata[[i]] ≠ {}, AppendTo[templist, Transpose[{Transpose[Vdata[[i]]][[
         airmassindex]], Transpose[Vdata[[i]]][[instmagindex]]}]],
     AppendTo[templist, {}]], {i, 1, Length[selected]}]];
  templist]
```

```
(*Gets a list of the coefficients from the linear fits of each object'
 s xy plot. If an object has been thrown out, it will not be added
 to the coefficient list. the format is {obj #, Rcoeff, Vcoeff}*)
coeffList[xyR_List, xyV_List, filterdata_List] := Module[{templist = {}},
  Do[If[Length[xyR[[i]]] > 1 && Length[xyV[[i]]] > 1, AppendTo[templist,
     {filterdata[[i, 1, objectindex]], Coefficient[Fit[xyR[[i]], {x, 1}, x], x],
      Coefficient[Fit[xyV[[i]], {x, 1}, x], x]}]], {i, 1, Length[xyR]}];
  templist]

(*Gets a list of r0 or v0 values from Rdata or Vdata
 and the corresponding average of their linear fit coefficients
 If the data object isn't an empty set, i.e. there is data for that object,
it will transpose the data list for that object and select the data
 for instrumental magnitude and airmass, and perform the calculation*)
naughtList[filterdata_List, Avg_] :=
 Module[{templist = {}, airmass, instmag, naught},
  Do[If[filterdata[[i]] ≠ {},
    airmass = Transpose[filterdata[[i]]][[airmassindex]];
    instmag = Transpose[filterdata[[i]]][[instmagindex]];
    naught = instmag - (Avg * airmass);
    AppendTo[templist, naught], AppendTo[templist, {}]],
   {i, 1, Length[filterdata]}];
  templist]

(*Takes Landolt values and creates a list of{object #, V mag, R mag}*)
createLandolt[inputlist_List] := Module[{templist = {}},
  (*Eliminates label row*)
  templist = Drop[inputlist, 1];
  (*Extracts just the objct, v mag, and r mag from each element*)
  Do[templist[[i]] = templist[[i]] /. {obj_, iden_, v_, both_, r_} → {obj, v, r},
   {i, 1, Length[templist]}];;
  templist]

(*Adjusts the values using the r0 or v0 and their
  corresponding Landolt values by using the equation R-r0*)
adjustLandolt[lValues_List, naught_List, filter_String] :=
 Module[{templist = {}, obj},
  Do[obj = lValues[[i, 1]];
   If[filter == "R", AppendTo[templist, lValues[[i, 3]] - naught[[obj]]],
    AppendTo[templist, lValues[[i, 2]] - naught[[obj]]]], {i, 1, Length[lValues]}];
  Do[PrependTo[templist[[i]], lValues[[i, 1]]], {i, 1, Length[lValues]}];
  templist]
```

```
(*Appends new information onto original
 Vdata and Rdata lists by transposing each object,
appending the new data, and returning it to its original format*)
addNew[filterData_List, naughtData_List, adjustedData_List] :=
 Module[{templistT = {}, tempAdjusted = {}, tempNaught = {}},
  tempAdjusted = adjustedData;
  (*Transposes each object and
   places that in the templist to be appended to later*)
  Do[If[Length[filterData[[i]]] > 1, AppendTo[templistT,
     Transpose[filterData[[i]]]]], {i, 1, Length[filterData]}];

  (*Adds naught data to tempNaught if it has useful data*)
  Do[If[Length[naughtData[[i]]] > 1, AppendTo[tempNaught, naughtData[[i]]]],
   {i, 1, Length[naughtData]}];

  (*Drops the object number from tempAdjusted list*)
  Do[tempAdjusted[[i]] = Drop[tempAdjusted[[i]], 1],
   {i, 1, Length[tempAdjusted]}];

  (*Appends new info (naught and adjusted) to the data list and returns it*)
  Do[AppendTo[templistT[[i]], tempNaught[[i]]];
   AppendTo[templistT[[i]], tempAdjusted[[i]]], {i, 1, Length[templistT]}];

  (*Re-Transposes each section of templistT to give usuful format*)
  Do[templistT[[i]] = Transpose[templistT[[i]]], {i, 1, Length[templistT]}];

  templistT
 ]

(*Joins Rdata and Vdata lists*)
dataJoin[Rlist_List, Vlist_List] := Module[{templist = {}},
  Do[templist = Join[templist, Rlist[[i]]], {i, 1, Length[Rlist]}];
  Do[templist = Join[templist, Vlist[[i]]], {i, 1, Length[Vlist]}];
  PrependTo[templist, titles];
  templist]

(*Extracts Ravg and Vavg from the adjusted values
 input lists are lists of lists. Each sublist is one object's data and
 the first element is the object number and the rest are data values
 create lists of the form {object,Ravg,Vavg} where
 each avg is average of Drop[list,1]*)
extract[rAdjusted_List, vAdjusted_List] :=
```

```
 Module[{templist = {}, newR = 0, newV = 0},
  Do[newR = Mean[Drop[rAdjusted[[i]], 1]];
   newV = Mean[Drop[vAdjusted[[i]], 1]];
   AppendTo[templist, {rAdjusted[[i, 1]], newR, newV}], {i, 1, Length[rAdjusted]}];
  PrependTo[templist, {"Object", "Adj. R Avg", "Adj. V Avg"}];
  templist]

(*Takes each objects r0 and v0 data,
Takes average v0 - r0 for each star and returns a
  list with {} for objects without more than one data point*)
rvSub[rMags_List, vMags_List] := Module[{minusList = {}},
  Do[If[Length[rMags[[i]]] > 1 && Length[vMags[[i]]] > 1,
    AppendTo[minusList, Mean[vMags[[i]]] - Mean[rMags[[i]]]],
    AppendTo[minusList, 0]], {i, 1, Length[rMags]}];
  minusList]

(*Gets μ and ξ for our field,
just one of each for the whole spreadsheet. Using equation V-R =
 μ(v0-r0) + ξ we make a plot using {x,y} pairs, one for each star where x is v0-
 r0 (averagev0 - averager0 found in rvSub) and y is Landolt V-R,
then get a linear fit and μ is the Coefficient term and ξ is the intercept
 Returns a list with {μ,ξ}*)
finalPairs[myLandolt_List, minData_List] :=
 Module[{myPlotList = {}, currentIndex = 0, finalFit, μ, ξ},
  Do[currentIndex = myLandolt[[i, 1]];
   AppendTo[myPlotList, {minData[[currentIndex]], myLandolt[[i, 4]]}],
   {i, 2, Length[myLandolt]}];
  finalFit = Fit[myPlotList, {x, 1}, x];
  μ = Coefficient[finalFit, x];
  ξ = Coefficient[finalFit, x, 0];
  {μ, ξ}]

(*Gets μ and ξ for the v filter data from equation V =
 v0 + μ(V-R) + ξ rearanded so V-v0 is on the left*)
vPairs[myLandolt_List, naughts_list] :=
 Module[{myPlotList = {}, currentIndex = 0, vFit, μ = 0, ξ = 0},
  Do[currentIndex = myLandolt[[i, 1]];
   Print[i];
   AppendTo[myPlotList,
    {myLandolt[[i, 4]], myLandolt[[i, 3]] - Mean[naughts[[currentIndex]]]}],
   {i, 2, Length[myLandolt]}];
  Print["Fitting"];
  vFit = Fit[myPlotList, {x, 1}, x];
```

```
  Print["Suceeded Fitting. Pulling coeffs."];
  μ = Coefficient[vFit, x];
  ξ = Coefficient[vFit, x, 0];
  {μ, ξ}]

(*Prevents you from overwriting an existing directory*)
safeCreateDirectory[dirName_] := Module[{newName, cutLoop},
  newName = dirName;
  cutLoop = False;
  (*If and While loops either get a valid directory name,
  or accept the overwrite*)
  SetDirectory[NotebookDirectory[]];
  If[DirectoryQ[newName], While[DirectoryQ[newName] && ! cutLoop, If[
     ChoiceDialog[StringForm["Directory `` Already Exists. Overwrite?", newName]],
     DeleteDirectory[NotebookDirectory[] <> newName, DeleteContents → True];
     cutLoop = True, newName = InputString["New File Directory Name: "]]]];
  CreateDirectory[NotebookDirectory[] <> "\\" <> newName];
  newName]

(*Prevents you from overwriting an existing directory. allows for an add-
 on to be passed in. The directory given by the user concatenated with the add-
 on will be checked. NOTE: only the original name, without the add-on,
is returned, allowing for files inside the folder to have simpler names*)
safeCreateDirectory[dirName_, addOn_] := Module[{newName, cutLoop},
  newName = dirName;
  cutLoop = False;
  (*If and While loops either get a valid directory name,
  or accept the overwrite*)
  SetDirectory[NotebookDirectory[]];
  If[DirectoryQ[newName <> addOn],
   While[DirectoryQ[newName <> addOn] && ! cutLoop, If[ChoiceDialog[
      StringForm["Directory `` Already Exists. Overwrite?", newName <> addOn]],
     DeleteDirectory[NotebookDirectory[] <> newName <> addOn,
      DeleteContents → True];
     cutLoop = True, newName = InputString["New File Directory Name: "]]]];
  CreateDirectory[NotebookDirectory[] <> "\\" <> newName <> addOn];
  newName]

(*END OF FUNCTION DECLARATIONS, BEGIN COMPUTATIONAL CODE*)

(*Split two lists, all objects in R filter, all objects in V filter*)
Rdata = dataSplit["R"];
Vdata = dataSplit["V"];
```

```
(*Creates two lists. xy pairs for all objects
 in R filter and xy pairs for all objects in V filter*)
Rxypairs = xyPairUp[Rdata];
Vxypairs = xyPairUp[Vdata];

(*Creates a list of coefficient lists of
 the form: {object #, R coefficient, V coefficient}*)
linearCoeff = coeffList[Rxypairs, Vxypairs, Rdata];

(*Gets the averages for each filter's linear fit coefficient*)
RCoeffAvg = Mean[Transpose[linearCoeff][[2]]];
VCoeffAvg = Mean[Transpose[linearCoeff][[3]]];

(*Calculates r0 and v0 values from the equation
  r0 = (Instrument magnitude) - (linear coefficient average)*(airmass)*)
r0 = naughtList[Rdata, RCoeffAvg];
v0 = naughtList[Vdata, VCoeffAvg];

(*Creates a list of the Landolt values we
 need in the form {obj #, Landolt R, Landolt V}*)
landoltValues = createLandolt[landoltData];

(*Adjusts the r0 and v0 values by taking R-r0*)
adjustedR = adjustLandolt[landoltValues, r0, "R"];
adjustedV = adjustLandolt[landoltValues, v0, "V"];

(*Creates list of Adjusted v -
 Adjusted r values for each object using lists above,
number is 0 if there was no (good) data for that star*)
vminr = rvSub[r0, v0];

(*Gets μ and ξ for the Standard Area*)
μξset = finalPairs[landoltData, vminr];

(*Gets μ and ξ for v filter values*)
vSet = Module[{myPlotList = {}, currentIndex = 0, vFit, μ = 0, ξ = 0},
   Do[currentIndex = landoltData[[i, 1]];
    AppendTo[myPlotList, {landoltData[[i, 4]], landoltData[[i, 3]] -
       Mean[v0[[currentIndex]]]}], {i, 2, Length[landoltData]}];
   vFit = Fit[myPlotList, {x, 1}, x];
   μ = Coefficient[vFit, x];
   ξ = Coefficient[vFit, x, 0];
```

```
    {μ, ξ}];

(*Combines all of the calculated data*)
fullR = addNew[Rdata, r0, adjustedR];
fullV = addNew[Vdata, v0, adjustedV];
combinedData = dataJoin[fullR, fullV];

(*Creates a data list for linear coefficients with average*)
coeffData = Prepend[linearCoeff, {"Object", "R", "V"}];
AppendTo[coeffData, {"Avg", RCoeffAvg, VCoeffAvg}];

(*Creates a data list for Adjusted R and Adjusted V values*)
summaryData = extract[adjustedR, adjustedV];


(*FILE WRITE FUNCTIONALITY BELOW*)

(*Combines all of the data into a list formatted solely for ease of reading*)
μInfo = {};
AppendTo[μInfo, Join[{"Mu_vr", "Zeta_vr"}, {, ,}, {"Mu_v", "Zeta_v"}]];
AppendTo[μInfo, Join[μξset, {, ,}, vSet]];
lastCombined = combinedData;
lastCombined[[1]] =
  Join[lastCombined[[1]], {, ,}, {"Coefficients"}, {, , ,}, {"Averages"}];
Do[lastCombined[[i + 1]] = Join[lastCombined[[i + 1]], {, ,}, coeffData[[i]]],
  {i, 1, Length[coeffData]}];
Do[lastCombined[[i + 1]] = Join[lastCombined[[i + 1]], {, }, summaryData[[i]]],
  {i, 1, Length[summaryData]}];
combineIndex = 5;
If[Length[coeffData] > Length[summaryData],
  combineIndex += Length[coeffData], combineIndex += Length[summaryData]];
Do[lastCombined[[i + combineIndex]] = Join[lastCombined[[i + combineIndex]],
    {, ,}, μInfo[[i]]], {i, 1, Length[μInfo]}];

(*Creates a time-tag add-on for the file name to
  distinguish between different nights of the same object*)
timeTag = dateConvert[dateTaken];

(*Creates a separate directory for each Standard area*)
writeLocation = InputString["Location to Write to: "];
writeLocation = safeCreateDirectory[writeLocation, timeTag];
origwriteLocation = writeLocation;
SetDirectory[NotebookDirectory[] <> "\\" <> origwriteLocation <> timeTag];
```

```
rawDataLocation = writeLocation <> "_Raw.csv";
landoltDataLocation = writeLocation <> "_Landolt.csv";
writeLocation = writeLocation <> ".csv";

(*Add feature to delete old file*)
Export[rawDataLocation, data];
Export[landoltDataLocation, landoltData];
Export[writeLocation, lastCombined];

SetDirectory[NotebookDirectory[]];
If[ChoiceDialog["Delete Original Data File?"], DeleteFile[location <> ".csv"]];
(*DeleteFile[location<>".csv"];*)
(*DeleteFile[landoltLocation<>".csv"];*)



(*Error Report Code*)
(*Check to see how close calculated values of μ and ξ get us to Landolt values*)
myVminR = {};
myV = {};
Do[AppendTo[myVminR, μξset[[1]] * vminr[[i]] + μξset[[2]]], {i, 1, Length[vminr]}];
Do[AppendTo[myV, Mean[v0[[i]]] + vSet[[1]] * myVminR[[i]] + vSet[[2]]],
  {i, 1, Length[vminr]}];
lanV = Transpose[Drop[landoltData, 1]][[3]];
lanR = Transpose[Drop[landoltData, 1]][[5]];
lanVminR = Transpose[Drop[landoltData, 1]][[4]];
vrDiff = myVminR - lanVminR;
vdiff = myV - lanV;
vrPercDiff = (Abs[myVminR - lanVminR] / lanVminR) * 100;
vPercDiff = (Abs[myV - lanV] / lanV) * 100;

differ = Transpose[
   {Join[{"Lan V-R"}, Transpose[Drop[landoltData, 1]][[4]]], Join[{"My V-R"},
     myVminR], Join[{"My-Lan V-R"}, vrDiff], Join[{"% off"}, vrPercDiff]}];

visual = Transpose[{Join[{"Lan V"}, Transpose[Drop[landoltData, 1]][[3]]],
    Join[{"My V"}, myV], Join[{"My-Lan V"}, vdiff], Join[{"% off"}, vPercDiff]}];

myR = myV - myVminR;
rPercDiff = (Abs[myR - lanR] / lanR) * 100;
rdiff = myR - lanR;
red = Transpose[{Join[{"Lan R"}, Transpose[Drop[landoltData, 1]][[5]]],
    Join[{"My R"}, myR], Join[{"My-Lan R"}, rdiff], Join[{"% off"}, rPercDiff]}];
```

```
(*Output error results to Error Log file*)
multi = {};
Do[AppendTo[multi, Join[differ[[i]], {""}, visual[[i]], {""}, red[[i]]]],
 {i, 1, Length[red]}]
output = {{"V-R difference avg = ", Mean[vrDiff]},
   {"V difference avg = ", Mean[vdiff]}, {"R difference avg = ", Mean[rdiff]},
   {"V-R difference std dev = ", StandardDeviation[vrDiff]},
   {"V difference std dev = ", StandardDeviation[vdiff]},
   {"R difference std dev = ", StandardDeviation[rdiff]},
   {"V-R avg % difference = ", Mean[vrPercDiff]}, {"V avg % difference = ",
    Mean[vPercDiff]}, {"R avg % difference = ", Mean[rPercDiff]}};
AppendTo[multi, {}];
Do[AppendTo[multi, output[[i]]], {i, 1, Length[output]}];

CreateDirectory[
  NotebookDirectory[] <> "\\" <> origwriteLocation <> timeTag <> "\\" <> "Error_Log"];
SetDirectory[NotebookDirectory[] <> "\\" <>
   origwriteLocation <> timeTag <> "\\" <> "Error_Log"];
Export["Error_Report.csv", multi];
Export["VminusR_Difference.jpg", ListPlot[vrDiff, PlotRange → All]];
Export["V_Difference.jpg", ListPlot[vdiff, PlotRange → All]];
Export["R_Difference.jpg", ListPlot[rdiff, PlotRange → All]];
(*END OF NORMAL CODE*)
Quit[]

(*OLD BASIC FILE WRITE IN CASE NEW FAILS
  (*Creates a separate directory for each Standard area*)
  writeLocation = InputString["Location to Write to: "];
CreateDirectory[writeLocation];
SetDirectory[NotebookDirectory[]<>"\\"<>writeLocation];
rawDataLocation = writeLocation<>"_Raw.csv";
landoltDataLocation = writeLocation<>"_Landolt.csv";
writeLocation = writeLocation<>".csv";

(*Add feature to delete old file*)
Export[rawDataLocation,data];
Export[landoltDataLocation,landoltData];
Export[writeLocation,lastCombined];
*)
```

```
(*View Error Report Right Now*)
differ // TableForm
visual // TableForm
red // TableForm
Print["V-R difference avg = ", Mean[vrDiff]]
Print["V difference avg = ", Mean[vdiff]]
Print["R difference avg = ", Mean[rdiff]]
Print["V-R difference std dev = ", StandardDeviation[vrDiff]]
Print["V difference std dev = ", StandardDeviation[vdiff]]
Print["R difference std dev = ", StandardDeviation[rdiff]]
Print["V-R avg % difference = ", Mean[vrPercDiff]]
Print["V avg % difference = ", Mean[vPercDiff]]
Print["R avg % difference = ", Mean[rPercDiff]]

ListPlot[vrDiff, PlotRange → All]
ListPlot[vdiff, PlotRange → All]
ListPlot[rdiff, PlotRange → All]

SetDirectory[NotebookDirectory[]];
```

Appendix C

Blazar Analysis and Reduction Code (BARC)

The code included is the Blazar class created in Python. An instance of this class holds all of the information gathered from a single blazar for all of its nights of observing.

The following function was included in the main.py file which read in all of the data. This function shows the process for reading in a single blazar.

```
#Loads a single blazar
def loadBlazar(sols, f_name=""):

    if "Data" not in f_name:
        path = "Data/"
    else:
        path =""
    if f_name == "":
        #Accepts .xls and .xlsx file types. If none specified, assume .xlsx
        f_name = input("File Name: ")

    temp = Blazar()
    status = temp.LoadData(path+f_name)                #Load in the data and do the error checks
```

```
    temp.LoadSolutions(sols)                                    #Add the solutions
    if status == 0:
        temp.NCCheck()                                          #Check any net count issues
        ss_comp = temp.AllStandardCompare(stop=True)            #Check our comparison stars and mark any bad ones
        if ss_comp == -1:
            print("All Stndard Stars Discarded")
            print("Skipping further calculations")
        else:
            if ss_comp == 1:
                print("More standard stars than allowed have been discarded. Proceed with caution")
            temp.NetCountCompare()
            if len(temp.solutions) == 0:
                print("No Solutions Nights for this Blazar, Ending Calculations")
            else:
                warning = temp.LoopAllSky()
                if warning == 0:
                    print("All programmed portions incoorperated")
                else:
                    print("Standard Deviation Within All Sky Solutions Exceeds %.2f, \
                    Proceed With Caution" %(.05))
                temp.Standardize()

            temp.Fit()

        print("\n")
        return temp
```

Shown below is the blazar class in which the calculations were performed.

```
# -*- coding: utf-8 -*-
"""
Created on Mon Jun  5 16:44:54 2017



@author: nvana
"""

from xlrd import open_workbook
import DataBase as db
import numpy as np
import os.path
from matplotlib import pyplot as plt

# Excel file Columns:
#    (0) Number
#    (1) Image File
#    (2) Object
#    (3) Name
#    (4) Magnitude
#    (5) Standard?
#    (6) Error
#    (7) Error(T)
#    (8) X
#    (9) Y
#    (10) Column
#    (11) Row
#    (12) Background
#    (13) S/N
#    (14) Mag Std
#    (15) Resid.
#    (16) Net Count
#    (17) Filter
#    (18) Date
#    (19) Time
#    (20) JD
```

```
#    (21) Airmass
#    (22) ExpTime
#    (23) Weight
#    (24) Notes
#    (25) Instrumental Magnitude (added in code below)

#index variables for info
#maybe a bit overkill, but it makes it easier to access and alter
#if I change the size or indices here, it changes it in the code
info_size = 17              #size of the 3rd dimension of info
Solution_index = 0
Vn_index = Solution_index + 1
Vnet_index = Vn_index + 1
Vstd_index = Vnet_index + 1
Vresult_index = Vstd_index + 1
Vupper_index = Vresult_index + 1
Vlower_index = Vupper_index + 1
Vinstr_index = Vlower_index + 1
Vcal_index = Vinstr_index + 1
Rn_index = Vcal_index + 1
Rnet_index = Rn_index + 1
Rstd_index = Rnet_index + 1
Rresult_index = Rstd_index + 1
Rupper_index = Rresult_index + 1
Rlower_index = Rupper_index + 1
Rinstr_index = Rlower_index + 1
Rcal_index = Rinstr_index + 1
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa

#Column indices for the information in data
#Because python data structures are zero-indexed, these are the column-1
img_col = 1
obj_col = 2
mag_col = 4
count_col = 16
filter_col = 17
date_col = 18
exposure_col = 22
```

```
instr_col = 24

#Column indices for the information in solutions
sol_num_col = 0                 #Column for the solution number
sol_date_col = sol_num_col + 1  #Column for the solution date
mu_vr_col = sol_date_col + 1    #Column for Mu_vr
zeta_vr_col = mu_vr_col + 1     #Column for Zeta_vr
mu_v_col = zeta_vr_col + 1      #Column for Mu_v
zeta_v_col = mu_v_col + 1       #Column for Zeta_v

class Blazar(object):

    """
    Blazar
    ======

    Provides:
        1. An object to hold the data
        2. Functions to calculate the important summaries
        3. Error checks to signal the user to any potential problems in calculations
        4. Graphical representation of the calculated data

    The data loaded into the blazar is photometry in the format given by Mira.
    Any other format may not function as expected and may cause errors.

    Sources of Error:
        1. Negative or zero net counts
        2. No All Sky Solutions
        3. Poor All Sky Solutions resulting in high standard deviation
        4. Too few objects (a missing aperature in Mira)
        5. Poor comparison stars resulting in too many being thrown out

    Format of solution list to be passed in
    ---------------------------------------

    Solution Number || Date || Mu_vr || Zeta_vr || Mu_v || Zeta_v

    There are several functions built in for the user (type Blazar.function?
    for more help):
```

```
    - Summarize() - Gives summary info for the blazar
    - ErrorSummary() - Gives a list of the errors encountered in calculation
    - ShowNetCount() - Displays a graph of the net count comparison
    - ShowStandardCompare() - Displays a graoh of each standard star compared to one
    - ShowRaw() - Displays the raw, uncalibrated data
    - ShowCalibrated() - Displays the final, standardized magnitudes of the blazar

"""

def __init__(self):
    self.file_loc = ""          #Name of file it was read from
    self.data = []              #Raw data read from the file
    self.ID = 0                 #Our given blazar number
    self.name = ""              #True name of blazar
    self.volatility = 0         #Volatility of blazar. 3 - widely varying, 0 - almost no variance
    self.Vpoly = None           #V polynomial equation
    self.Rpoly = None           #R polynomial equation
    self.Vpoly_r_sqr = 0        #R^2 value of the 3rd order polynomial fit in V
    self.Rpoly_r_sqr = 0        #R^2 value of the 3rd order polynomial fit in R
    self.dates = []             #List of dates in data
    self.comparison_index = 1   #Index to compare standards to (index 1 = star 2)
    self.objects = 0            #Number of objects

    #First dimension is object (off by one, i.e., 0 is object 1, but object 1 will be left empty)
    #Second dimension is as follows
    #   (0) - V magnitude
    #   (1) - V std dev
    #   (2) - R magnitude
    #   (3) - R std dev
    self.standard_magnitudes = []       #Magnitudes of the secondary standard stars
    self.discarded = []                 #Discarded standard stars

    # First dimension is the night (NightList)
    # Second dimension is the object (off by one, i.e., 0 is object 1)
    # Third dimension is this:
    #   (0) Solutions - Number of All Sky Solutions for that night (only place on object 1/index 0)
    #   (1) Vn - Number of data points
    #   (2) Vnet - Sum of counts
```

```
#   (3) Vstd - Standard deviation of data points (Poisson)
#   (4) Vresult - Result of -2.5*log10(net_count_1/net_count_2)
#       for blazar: top object 1 & bottom = total of all others
#   (5) Vupper - difference between (3) and (3) with the Vstd added to net_count_1
#   (6) Vlower - same as (4) but with Vstd subtracted
#   (7) Vinstr - Instrumental V magnitude average (-2.5*log10(Net Count/Exposure Time))
#   (8) Vcal - Calibrated V magnitude average for blazar and objects, average offset for
#       objects on a night without an all sky solution
#   (9) Rn - Number of data points
#   (10) Rnet - Sum of counts
#   (11) Rstd - Standard deviation of data points (Poisson)
#   (12) Rresult - Same as Vresult but in R filter
#   (13) Rupper - Difference between Rresult and the upper bound based on poisson std dev
#   (14) Rlower - Difference between Rresult and the lower bound based on poisson std dev
#   (15) Rinstr - Instrumental R magnitude average
#   (16) Rcal - Calibrated R magnitude average for blazar and objects, average offset for
#       objects on a night without an all sky solution
self.info = []                      #Data following format above

#First dimension is each solution
#Second dimension is as follows:
#   (0) - Solution Number
#   (1) - Date
#   (2) - Mu_vr
#   (3) - Zeta_vr
#   (4) - Mu_v
#   (5) - Zeta_v
#   (6) - Area Name
self.solutions = []                 #Solutions for this blazar

#Error Variables
#e_list (error list) is as follows: (0 - no issue, 1 - issue)
#   (0) - Too few objects on some nights
#   (1) - Faint Blazar, too few net counts on blazar
#   (2) - General net count issue, at least one night has issues.
#       value of 2 = issue resolved (all bad net count stars were thrown out)
#   (3) - No remaining stars = -1, too many stars flagged = 1, fine = 0
#   (4) - Too many solutions flagged as bad
```

```
        #   (5) - Column or element issues
        self.e_list = np.zeros((6))          #Array of the types of errors
        self.deficient_obj = []              #Nights with too few objects
        self.ncError = []                    #List of nights with errors
        self.blError = []                    #List of nights with net count errors for the blazar
        self.nc_obj = []                     #List of objects with bad net counts

    #Gives summary information of the blazar
    def Summarize(self):
        """
        Summarize
        =========

        Displays some basic summary information for the blazar

        """
        print("Blazar %d" %self.ID)
        print("Identifier: " + self.name)
        print("Volatility: %d" %self.volatility)
        print("Observed Nights: %d" %len(self.dates))
        print("3rd Order Polynomial R^2 (V): %.4f" %(self.Vpoly_r_sqr))
        print("3rd Order Polynomial R^2 (R): %.4f" %(self.Rpoly_r_sqr))

    #Gets the name of the blazar from the file name using the convention of naming it bl##
    def PullID(self, f_block):
        i = 0
        temp_id = ""
        while f_block[i] != '.':
            if f_block[i].isdigit():
                temp_id += f_block[i]

            i += 1

        self.ID = int(temp_id)

    #loads data from the file name
    def LoadData(self, f_name):
```

```
#If there's no dot, there's no file extension
if not "." in f_name:
    #Assume a .xlsx file
    print("No Explicit File Type Given. Assuming . xlsx Format")
    f_name += ".xlsx"

if(os.path.isfile(f_name)):
    self.file_loc = f_name
    print("Reading from " + f_name)
    wb = open_workbook(f_name)
    count = 0
    #For each sheet in the Excel file
    for s in wb.sheets():
        #For each row on that sheet
        for row in range(s.nrows):
            #This avoids adding extra title rows. Only add the title row the first time
            if not (count != 0 and s.cell(row,0).value == "#"):
                col_value = []
                #For each column in that row
                for col in range(s.ncols):
                    #Grab the value on that sheet at that specific cell
                    value  = (s.cell(row,col).value)
                    col_value.append(value)
                self.data.append(col_value)
            count += 1
    self.PullID(self.data[1][img_col])              #Find the Blazar number from the image name

    #populate dates list
    for row in self.data:
        if row[date_col] not in self.dates and not isinstance(row[date_col], str):
            self.dates.append(row[date_col])

    self.ObjectMismatch()                           #First error function, also set self.objects
    self.CalcInstrumentalMagnitude()                #Calculate instrumental magnitude from the data
    self.LoadInfo()                                 #Split the data into useful elements
    return 0            #It succeeded
else:
    print(f_name + " does not exist")
```

```
        return -1           #No File, do not continue

#Takes the information from data and breaks it into a useful organization
def LoadInfo(self):

    #Make info into a 3D array of data
    #First dimension as large as dates
    #Second dimension as large as largest object number
    #Third dimension as large as number of criteria indicated in __init__
    self.info = np.zeros((len(self.dates),self.objects,info_size))

    #project and select to pull the data values I need
    for i in range(len(self.dates)):
        for j in range(self.objects):

            #V filter
            #Gather Vn
            self.info[i][j][Vn_index] = \
                        len(db.project(db.select(self.data,[(date_col,self.dates[i]),\
                                (obj_col,j+1),(filter_col,"V")]),[obj_col]))
            #Gather net counts
            self.info[i][j][Vnet_index] = \
                        np.nansum(db.project(db.select(self.data,[(date_col,self.dates[i]), \
                                    (obj_col,j+1),(filter_col,"V")]),[count_col]))
            if self.info[i][j][Vnet_index] > 0:
                #sqrt of net counts or poisson standard deviation
                self.info[i][j][Vstd_index] = np.sqrt(self.info[i][j][Vnet_index])
            #Gather V instrumental magnitude average
            self.info[i][j][Vinstr_index] = \
                    np.nanmean(db.project(db.select(db.select(self.data, \
                        [(date_col,self.dates[i]),(obj_col,j+1),(filter_col,"V")]), \
                        [(instr_col,"INV!")],mirror=True),[instr_col]))

            #R filter
            #Gather Rn
            self.info[i][j][Rn_index] = \
                    len(db.project(db.select(self.data,[(date_col,self.dates[i]),\
                        (obj_col,j+1),(filter_col,"R")]),[obj_col]))
```

```
            #Gather net counts
            self.info[i][j][Rnet_index] = \
            np.nansum(db.project(db.select(self.data,[(date_col,self.dates[i]),(obj_col,j+1), \
                        (filter_col,"R")]),[count_col]))
            if self.info[i][j][Rnet_index] > 0:
                self.info[i][j][Rstd_index] = np.sqrt(self.info[i][j][Rnet_index])
            #Gather R instrumental magnitdue average
            self.info[i][j][Rinstr_index] = \
            np.nanmean(db.project(db.select(db.select(self.data, \
                [(date_col,self.dates[i]),(obj_col,j+1),(filter_col,"R")]),[(instr_col,"INV!")], \
                mirror=True),[instr_col]))

def LoadSolutions(self, values = []):
    if values != []:
        self.solutions = []
        for row in values:
            #Check to see if the date for the solutions was a night we observed this blazar
            if row[1] in self.dates:
                self.solutions.append(row)

#Calculates the instrumental magnitude for each point and appends it to the row in data
def CalcInstrumentalMagnitude(self):
    i = 0
    for row in self.data:
        if i == 0:
            #Append the title
            if len(row) < 25:
                row.append("Instrumental Magnitude")
            else:
                row[instr_col] = "Instrumental Magnitude"
        elif isinstance(row[count_col], str) or isinstance(row[exposure_col], str):
            self.e_list[5] = 1
        elif row[count_col] > 0 and row[exposure_col] > 0:
            #Append the instrumental magnitude onto the row
            if len(row) < 25:
                row.append(-2.5*np.log10(row[count_col]/row[exposure_col]))
            else:
                row[instr_col] = -2.5*np.log10(row[count_col]/row[exposure_col])
```

```
            else:
                #Append a warning label to tell the user the number cannot be calculated
                if len(row) < 25:
                    row.append("INV!")          #Could be cause by net count <= 0 or exposure time = 0
                else:
                    row[instr_col] = "INV!"
            i += 1
        if self.e_list[5] == 1:
            #Two possible fixes:
            #Delete empy rows after last row of data on each worksheet (for some reason that works)
            #Shift columns over (They may have been shifted in copying)
            print("Data has balnk rows and/or shifted columns")
            return -1
        else:
            return 0

    #Compares each standard star to one of the others to see if any of our chosen objects is bad
    def SingleStandardCompare(self):
        #for each night observed
        for night in self.info:
            #for every object observed that night from 2 on (index 1 and on)
            for obj in range(len(night))[1:]:

                #net counts for the current and comparison objects
                Vnet_compare = night[self.comparison_index][Vnet_index]
                Vnet_obj = night[obj][Vnet_index]
                Rnet_compare = night[self.comparison_index][Rnet_index]
                Rnet_obj = night[obj][Rnet_index]

                #if the Vnet on both the comparison star and the current standard are positive
                if Vnet_compare > 0 and Vnet_obj > 0:

                    #pull and calculate values from elements already in array
                    Vresult = -2.5*np.log10(Vnet_compare/Vnet_obj)
                    Vstd = night[self.comparison_index][Vstd_index]

                    #place results in array
                    night[obj][Vresult_index] = Vresult
```

```
                night[obj][Vupper_index] = \
                    abs(Vresult - -2.5*np.log10((Vnet_compare + Vstd)/Vnet_obj))
                night[obj][Vlower_index] =\
                    abs(Vresult - -2.5*np.log10((Vnet_compare - Vstd)/Vnet_obj))

            if Rnet_compare > 0 and Rnet_obj > 0:

                #pull and calculate values from elements already in array in R filter now
                Rresult = -2.5*np.log10(Rnet_compare/Rnet_obj)
                Rstd = night[self.comparison_index][Rstd_index]

                #place R results in array
                night[obj][Rresult_index] = Rresult
                night[obj][Rupper_index] = \
                    abs(Rresult - -2.5*np.log10((Rnet_compare + Rstd)/Rnet_obj))
                night[obj][Rlower_index] = \
                    abs(Rresult - -2.5*np.log10((Rnet_compare - Rstd)/Rnet_obj))

#Helper function.
#Creates the list of thrown stars giving absolute precedence to stars with a value of nan
#Takes most of its variables from AllStandardCompare()
def _Discard(self, threshold, max_toss, stop):
    hard_toss = []                      #List of nan stars, absolutely MUST be discarded
    soft_toss = []                      #List of bad stars other than nan, should be discarded

    #Make list of tossed stars
    for obj in range(len(self.info[0]))[1:]:
        #These are the criteria I'm looking at
        vstd = np.nanstd(self.info[:,obj,Vresult_index])
        rstd = np.nanstd(self.info[:,obj,Rresult_index])

        #Append if either has a nan value
        if np.isnan(vstd) or np.isnan(rstd):
            hard_toss.append(obj)
        #Append if the standard deviation in either filter breaks the threshold
        elif vstd > threshold or rstd > threshold:
            soft_toss.append(obj)
```

```
        #Regardless of the following cases, hard_toss stars WILL be discarded
        self.discarded = hard_toss

        #Possibilities from here on:
        #   hard_toss and soft_toss could either be <, >, or == to max_toss, or all of the stars
        #   (4 cases each)
        #   stop = true or false, doubling the cases
        #   regardless, discarded is hard, in >= max or == all cases, nothing will change for stop
        #   for last stop case and all non-stop cases, soft_toss will be added to hard_toss as permitted
        #   from there, if self.discarded contains every star, set flag to -1, meaning stop calculations
        #   otherwise, if more stars than max_toss were flagged as bad, falg 1 to say watch out
        #   otherwise, it's fine, flag 0, then return the flag, whatever it is

        if stop:
            for obj in soft_toss:
                if len(self.discarded) < max_toss:
                    self.discarded.append(obj)
        else:
            #Place all of the soft_toss stars into discarded
            self.discarded[:0] = soft_toss

        #Now set flags
        if len(self.discarded) == self.objects:
            self.e_list[3] = -1                             #No remaining stars
        elif len(hard_toss) + len(soft_toss) > max_toss:
            self.e_list[3] = 1                              #Remaining stars, but errors exceed limits
        #Otherwise, flag is 0 by default

        return self.e_list[3]                               #Return whatever value was in the flag

    #Perform singleStandardCompare until fewer than the threshold number of objects have been thrown out,
    #or until all objects have been used as the comparison index and the best one chosen
    #threshold is the standard deviation above which I will decide to toss the stars
    #Setting stop to true will stop throwing out stars once it has hit the max_toss limit
    #(really it just limits the final list)
    def AllStandardCompare(self, threshold=.05, stop=False):

        #Reset discarded list to empty
```

```
self.discarded = []

#Greater than 1 because standard deviation calculation needs minimun 2 points
if(len(self.info) > 0):
    #Don't throw away more than this many stars, varies depending on number of objects
    max_toss = int((len(self.info[0]) - 1)/2)
    best = {}               #Dictionary from number of tossed stars to comparison_index

    #For each object on the first night observed
    #(assume this will be the same every night) excluding object 1 (blazar)
    for i in range(len(self.info[0]))[1:]:
        tossed = 0          #Number of stars tossed for current comparison_index

        #Perform the standard comparison on the current comparison_index
        self.SingleStandardCompare()

        #Check to see how many stars are tossed given this criteria
        for j in range(len(self.info[0]))[1:]:
            #For ease of read, these are the criteria I'm looking at
            vstd = np.nanstd(self.info[:,j,Vresult_index])
            rstd = np.nanstd(self.info[:,j,Rresult_index])
            #if either the V or R filter breaks the threshold standard deviation
            #or if either is np.nan
            if vstd > threshold or rstd > threshold or np.isnan(vstd) or np.isnan(rstd):
                tossed += 1

        #if the current number tossed has not already been encountered
        if not tossed in best:
            #The number of stars tossed will be attributed to the current comparison index
            best[tossed] = self.comparison_index

        #if the current number tossed is below the max allowed, exit the first for loop
        if tossed <= max_toss:
            break
        else:
            #increment the comparison_index
            if self.comparison_index < len(self.info[0]) - 1:
                self.comparison_index += 1
```

```
                #if the index is already on the last object, return to star 2 (index 1)
                else:
                    self.comparison_index = 1

        #After exiting for loop, redo singleStandardCompare() with the best comparison_index
        #comparison_index is whichever had fewest thrown stars
        self.comparison_index = best[min(best)]
        self.SingleStandardCompare()

        return self._Discard(threshold, max_toss, stop)             #Return the result

#Compare the net counts of the blazar to the net counts of the remaining stars combined
#Once compared, calculate volatility
#Very similar to SingleStandardCompare except it always compares index 1 to combination of all
def NetCountCompare(self):
    #Go through each night
    for night in self.info:
        Vconglomerate = 0           #The "pile" of net counts from the other standards in V filter
        Rconglomerate = 0           #Same as Vcongolmeratebut in the R filter

        #Gather the net counts that are not tossed
        for obj in range(len(night))[1:]:
            #Add if not included in discard list
            if obj not in self.discarded:
                Vconglomerate += night[obj][Vnet_index]
                Rconglomerate += night[obj][Rnet_index]

        #Calculate the results and upper/lower limits
        Vnet_bl = night[0][Vnet_index]                  #Net counts for the blazar in V
        Rnet_bl = night[0][Rnet_index]                  #Net counts for the blazar in R

        #Only calculate if the numbers are above 0
        if Vnet_bl > 0 and Vconglomerate > 0:

            #Pull and calculate values to place later
            Vresult = -2.5*np.log10(Vnet_bl/Vconglomerate)
            Vstd = night[0][Vstd_index]
```

```
            #Place values in array
            night[0][Vresult_index] = Vresult
            night[0][Vupper_index] = abs(Vresult - 2.5*np.log10((Vnet_bl + Vstd)/Vconglomerate))
            night[0][Vlower_index] = abs(Vresult - 2.5*np.log10((Vnet_bl - Vstd)/Vconglomerate))

        if Rnet_bl > 0 and Rconglomerate > 0:

            #Pull and calculate values to place later
            Rresult = -2.5*np.log10(Rnet_bl/Rconglomerate)
            Rstd = night[0][Rstd_index]

            #Place values in array
            night[0][Rresult_index] = Rresult
            night[0][Rupper_index] = abs(Rresult - 2.5*np.log10((Rnet_bl + Rstd)/Rconglomerate))
            night[0][Rlower_index] = abs(Rresult - 2.5*np.log10((Rnet_bl - Rstd)/Rconglomerate))

    #Now, calculate how volatile the blazar is/how much it varies
    #Assign a number based on the following:
    #std dev in Vresult/Rresult / std dev in (Vresult - Rresult)
    #3 -> [3,infinity), 2 -> [2,3), 1 -> [1,2), 0 -> (-infinity,1)
    #A higher number means that it is less likely that the change is due to uncertainty
    #in the difference between the filters
    #We assume that flaring will occure roughly equally in R as in V

    #std dev of Vresult for object 1/index 0 (blazar)
    Vres_std = np.nanstd(self.info[:,0,Vresult_index])
    #std dev of Rresult for object 1/index 0 (blazar)
    Rres_std = np.nanstd(self.info[:,0,Rresult_index])
    #std dev for Vresult - Rresult
    VminR_std = np.nanstd(self.info[:,0,Vresult_index] - self.info[:,0,Rresult_index])

    Vvol = 0
    Rvol = 0

    if VminR_std > 0:
        Vvol = Vres_std/VminR_std                    #V volatility
        Rvol = Rres_std/VminR_std                    #R volatility
```

```
    if Vvol >= 3 or Rvol >= 3:
        self.volatility = 3
    elif Vvol >= 2 or Rvol >= 2:
        self.volatility = 2
    elif Vvol >= 1 or Rvol >= 1:
        self.volatility = 1
    else:
        self.volatility = 0

#Calibrates the magnitudes based on the All Sky Solutions
#A single run through assuming the solutions are good, a later function will repeatedly call this
#until the values have stabilized
def AllSkySolution(self):

    #Reset pre-existing values
    self.standard_magnitudes = np.zeros((self.objects,4))
    self.info[:,:,Solution_index] = 0
    self.info[:,:,Vcal_index] = 0
    self.info[:,:,Rcal_index] = 0

    #Values will hold the solution results for each object
    #First dimension is the object
    #Second dimension is the solution
    #Third Dimension is as follows:
    #   (0) - Solution Number
    #   (1) - Calibrated V
    #   (2) - Calibrated R
    values = np.zeros((self.objects,len(self.solutions),3))

    loop = 0                    #Loop counter for current solution

    #Do each solution
    for sol in self.solutions:

        index = 0                   #The index in self.dates corresponding to this solution
                                    #also the access index for self.info
        mu_vr = sol[mu_vr_col]      #Mu_vr
        zeta_vr = sol[zeta_vr_col]  #Zeta_vr
```

```
mu_v = sol[mu_v_col]          #Mu_v
zeta_v = sol[zeta_v_col]      #Zeta_v

#Find the index for the corresponding date
for i in range(len(self.dates)):
    #Found the corresponding date
    if self.dates[i] == sol[sol_date_col]:
        index = i
        break

#Loop through info at the night of the solution
for obj in range(len(self.info[index])):
    #Equations are as follows:
    #(Capital V and R are true values, lowercase are instrumental magnitudes)
    #V-R = Mu_vr*(v-r) + Zeta_vr
    #V = v + Mu_v*(V-R) + Zeta_v
    #R = V - (V-R)
    VminR = mu_vr*(self.info[index][obj][Vinstr_index] - \
                        self.info[index][obj][Rinstr_index]) + zeta_vr
    V = self.info[index][obj][Vinstr_index] + (mu_v*VminR) + zeta_v
    R = V - VminR

    #Place at appropriate spot in values array
    values[obj][loop][0] = sol[0]          #Place the solution number
    values[obj][loop][1] = V               #Place V magnitude
    values[obj][loop][2] = R               #Place R magnitude

    #Place in self.info as well
    #Because there can be multiple solutions per night, use the following formula to
    #keep the average:
    #Vcal = (V + (Vcal*Solutions))/(Solutions + 1)
    #This will weight the current average to how many points contributed
    self.info[index][obj][Vcal_index] = \
            (V + (self.info[index][obj][Vcal_index] * \
            self.info[index][obj][Solution_index]))/(self.info[index][obj][Solution_index] \
            + 1)
    self.info[index][obj][Rcal_index] = \
            (R + (self.info[index][obj][Rcal_index] * \
```

```
                    self.info[index][obj][Solution_index]))/(self.info[index][obj][Solution_index] \
                    + 1)            #Follow formula above
                self.info[index][obj][Solution_index] += 1              #Increment Solution count

            loop += 1

        #Now values is populated, populate standard_magnitudes
        for obj in range(len(self.standard_magnitudes))[1:]:
            #V magnitude is the average V through all solutions on that object
            self.standard_magnitudes[obj][0] = np.nanmean(values[obj,:,1])
            #V standard deviation is using V on all solutions for that object
            self.standard_magnitudes[obj][1] = np.nanstd(values[obj,:,1])
            #R magnitude is the average R through all solutions on that object
            self.standard_magnitudes[obj][2] = np.nanmean(values[obj,:,2])
            #R standard deviation is using R on all solutions for that object
            self.standard_magnitudes[obj][3] = np.nanstd(values[obj,:,2])

        return values

    #Support function for finding standard deviation of all elements except one
    #Create a copy of the list except the excluded item, return std dev
    def stdex(self, table, index):
        copy = []
        for i in range(len(table)):
            if i != index:
                copy.append(table[i])

        return np.nanstd(copy)

    #Support function for finding the outlier solution to toss
    #Pass in a 3D array
    def FindWorst(self, table, threshold=.05):
        worst = {0:0}       #Dictionary from the solution number to its frequency of being the outlier
        #Iterate through each object and find which solution is the worst for it
        for obj in table:
            worst_sol = 0                   #Solution number of the worst solution for current object
            std_total = 0                   #Combination of the R and V std dev for the worst solution
```

```
        #Iterate through solutions and find which one is the worst and above threshold
        for i in range(len(obj)):
            if not i in range(self.objects):
                Vtemp = self.stdex(obj[:,1],i)          #Std dev of V without solution at i
                Rtemp = self.stdex(obj[:,2],i)          #Std dev of R without solution at i

                #If either is above the threshold and worst than the current worst
                if (Vtemp > threshold or Rtemp > threshold) and (Vtemp + Rtemp) > std_total:
                    worst_sol = obj[i,0]
                    std_total = Vtemp + Rtemp

                #Add to map or increment the worst count if it already exists
                if worst_sol in worst:
                    worst[worst_sol] += 1
                else:
                    worst[worst_sol] = 1

    #Finds the key that maps to the largest number in the dictionary
    worst_key = max(worst, key=lambda key:worst[key])   #The key with the most counts of "worst"
    return worst_key                                    #Returns 0 if all are below threshold

#Do AllSkySolution() repeatedly and throw away the worst solution each time until
#There are too few nights to throw, or the standard deviations fall within the threshold
#Threshold is the standard deviation in the V and R magnitudes above which we say there's a
#bad solution
def LoopAllSky(self, threshold =.05, manual_min=0):
    #First, call AllSkySolution and pull the values list from it
    values = self.AllSkySolution()

    min_sols = int(len(self.solutions)/2)           #Leave at least half of the solutions
    if min_sols < 3:
        #No fewer than 3 solutions
        min_sols = 3

    #If there are only 3 solutions, allow a single one to be thrown out
    if len(self.solutions) == 3:
        min_sols = 2
```

```
#The user can manually decide how many solutions to allow
if manual_min > 0:
    min_sols = manual_min

worst_sol = -1              #Dummy value, allows entering while loop

#Loop through and throw away bad solutions until the standard deviations fall
#within the threshold
#Or there are too few to throw out
while len(self.solutions) > min_sols and worst_sol != 0:

    #Finds the worst solution. If all are within the threshold, worst_sol == 0
    worst_sol = self.FindWorst(values)

    #If there is a worst solution, throw it out and re-run AllSkySolution
    if worst_sol != 0:

        index = -1          #Dummy value, will become the index in solutions to delete

        #Run through the solutions
        for i in range(len(self.solutions)):
            #Once you find the solution...
            if self.solutions[i][0] == worst_sol:
                index = i                           #Assign the index
                break                               #Exit the for loop

        #Eliminate the solution at the found index. Check that it is a valid index
        if index in range(len(self.solutions)):
            del self.solutions[index]
            values = self.AllSkySolution()          #Run AllSkySolution with new solutions

#while loop ended, either we have too few solutions to continue, or they are fine
if worst_sol != 0:
    self.e_list[4] = 1      #Place the flag for too many solutions tossed
    return -1               #Warning, the data may not be as accurate as possible
else:
    return 0                #Ideal situation
```

```
#Now that we have the secondary standard stars' magnitudes, standardize the blazar
def Standardize(self):
    #The same number for most stars allowed to be discarded
    max_toss = int((self.objects - 1)/2)

    #If there are fewer stars in self.discarded than max_toss, throw out the faintest to
    #improve data
    while len(self.discarded) < max_toss:
        dimmest = -1000             #Magnitude of the current faintest star, more positive is dimmer
        star = 0                    #Star index

        #Loop through objects in self.standard_magnitudes
        for obj in range(len(self.standard_magnitudes))[1:]:
            #Only consider if it is not in self.dicarded
            if not obj in self.discarded:
                #Combine its V and R magnitudes and compare to current dimmest
                #Combined magnitude
                mag = self.standard_magnitudes[obj,1] + self.standard_magnitudes[obj,2]
                #If it is dimmer, change the current dimmest
                if mag > dimmest:
                    star = obj
                    dimmest = mag

        #Now we have the dimmest star
        #Add if it's a real star that isn't already in the list
        if star != 0 and not star in self.discarded:
            self.discarded.append(star)
        #Otherwise, something went wrong, so break to avoid an infinite loop
        else:
            break

    #Find out if throwing away stars threw away all our bad net count issues
    #If we find an object in self.nc_obj that has not been thrown out, set to false
    fixed = True
    for obj in self.nc_obj:
        if obj not in self.discarded:
            #We found one that did not get thrown. Likely due to too many bad standard stars
            fixed = False
```

```
#If we fixed the issue, set the new error flag to tell the user there are net count issues,
#But that they have been ignored
if fixed:
    self.e_list[2] = 2          #Set the unique error flag

#Loop through all nights in self.info
for night in self.info:
    #Only calculate if it was not an All Sky Solution night
    if night[0,Solution_index] == 0:

        total_Voffset = 0       #The total offset in V (true magnitude - instrumental magnitude)
        Vcount = 0              #How many stars were factored in (were not discarded)
        total_Roffset = 0       #Total offset in R
        Rcount = 0              #Count in R

        #Loop through all of the objects other than the blazar and those that have been discarded
        for obj in range(len(night))[1:]:
            #Only if it was not thrown away
            if not obj in self.discarded:
                #find the offset and add it to the total to average later
                if night[obj,Vinstr_index] != 0:
                    Voffset = self.standard_magnitudes[obj,0] - night[obj,Vinstr_index]
                    Vcount += 1
                    total_Voffset += Voffset
                if night[obj,Rinstr_index] != 0:
                    Roffset = self.standard_magnitudes[obj,2] - night[obj,Rinstr_index]
                    Rcount += 1
                    total_Roffset += Roffset

        #Calculate the averages
        Vavg_offset = total_Voffset/Vcount
        Ravg_offset = total_Roffset/Rcount

        #Add the average offsets to the blazar instrumental magnitudes and store
        night[0,Vcal_index] = Vavg_offset + night[0,Vinstr_index]
        night[0,Rcal_index] = Ravg_offset + night[0,Rinstr_index]
```

```
#Finds the 3rd order polynomial fit to the net count data and calculates the R^2 value
def Fit(self):
    #Get ordered lists to use for equations
    order = np.argsort(self.dates)                          #Make an array of the reordering
    dates_s = np.array(self.dates)[order]                   #Order the dates
    v_s = np.array(self.info[:,0,Vresult_index])[order]     #Order the V results
    r_s = np.array(self.info[:,0,Rresult_index])[order]     #Order the R results

    #Use the V and R results to get the 3rd order polynomial fit
    self.Vpoly = np.poly1d(np.polyfit(dates_s, v_s, 3))
    self.Rpoly = np.poly1d(np.polyfit(dates_s, r_s, 3))

    #R^2 is 1 - ( sum((actual - predicted)^2)/sum((actual - mean)^2) )
    Vavg = np.nanmean(v_s)                                  #Average V values
    Ravg = np.nanmean(r_s)                                  #Average R values

    #Get the upper portion of the equation: sum( (actual - predicted)^2 )
    #Residual sum of squares
    Vlist = []                  #List for storing the (actual-predicted)^2
    Rlist = []
    for index in range(len(dates_s)):
        Vlist.append((v_s[index] - self.Vpoly(dates_s[index]))**2)
        Rlist.append((r_s[index] - self.Rpoly(dates_s[index]))**2)

    #Get the sum
    Vres = np.nansum(Vlist)             #V residual
    Rres = np.nansum(Rlist)             #R residual

    #Get the lower portion of the equation: sum( (actual - mean)^2 )
    Vtot = np.nansum([(x - Vavg)**2 for x in v_s])      #V total sum of squares
    Rtot = np.nansum([(x - Ravg)**2 for x in r_s])      #R total sum of squares

    #If Vtot or Rtot are 0, there was an issue
    if Vtot != 0:
        self.Vpoly_r_sqr = 1 - (Vres/Vtot)              #Finally, the R^2 value for V
    if Rtot != 0:
        self.Rpoly_r_sqr = 1 - (Rres/Rtot)              #Finally, the R^2 value for R
```

```
##########################################################################################################
#GRAPH FUNCTIONS##########################################################################################
##########################################################################################################

    #Shows a plot of the V and R Net Count comparison for the blazar object
    #save being set to true will save a copy of the plot and px is the dpi size of the plot
    #If you set show to false, it will not show the plot, useful for saving only
    def ShowNetCount(self, show=True, save=False, px=150):
        """
        ShowNetCount
        ============

        Shows the net counts of the blazar compared to the combined net counts of the comparison
        stars (those not discarded) using the equation: -2.5*log10(bl_net_count/star_net_count)

        Optional Variables:

            show:
                Default = True
                Set to False to suppress displaying the graph

            save:
                Default = False
                Set to True to save a copy of the graph

            px:
                Default = 150
                Alter to adjust the size of the graph
                Used for the dpi optional variable on pyplot

        """
        fig = plt.figure(dpi=px)                            #figure to add axes to
        ax = fig.add_subplot(111)                           #the axes/plot to lay on to
        ax.set_title("Bl%d Net Counts" %(self.ID))          #Set the title

        #Sort the dates and assign an order to the rearranging
        order = np.argsort(self.dates)
        dates_s = np.array(self.dates)[order]                       #Create a sorted date list
```

```
        vs = np.array(self.info[:,0,Vresult_index])[order]          #Create a sorted array of Vresult
        rs = np.array(self.info[:,0,Rresult_index])[order]          #Create a sorted array of Rresult

        ax.plot(dates_s, vs, "-go", label = "Delta V")      #Plot the v data - color = green
        ax.plot(dates_s, rs, "-ro", label = "Delta R")      #Plot the r data - color = red

        plt.gca().invert_yaxis()                            #Invert the y axis
        plt.legend(loc="upper right")
        if show:
            plt.show()

        #Save a copy of the plot
        if save:
            fig.savefig("Plots/Bl%d_NetCount.png" %(self.ID))

    #Shows a plot of standard star comparisons
    #saveV or saveR being set to true will save a copy of the plots and px is the dpi size of the plot
    #If you set show to false, it will not show the plot, useful for saving only
    def ShowStandardCompare(self, show=True, saveV=False, saveR=False, px=150):
        """
        ShowStandardCompare()
        =====================

        Shows the comparison of all of our secondary standard stars. Each star will be compared
        to one other. You can see the result of -2.5*log10(comparison_net_count/other_net_count)
        for each star. Each filter gets its own graph with all comparison stars plotted on each.

        Optional Variables:

            show:
                Default = True
                Set to False to suppress displaying the graph

            saveV:
                Default = False
                Set to True to save a copy of the V filter graph

            saveR:
```

```
        Default = False
        Set to Trueto save a copy of the R filter graph

    px:
        Default = 150
        Alter to adjust the size of the graph
        Used for the dpi optional variable on pyplot

"""

figV = plt.figure(1, dpi=px)        #figure to add V to
figR = plt.figure(2, dpi=px)        #figure to add R to
v_ax = figV.add_subplot(111)        #the axis to lay V values on to
r_ax = figR.add_subplot(111)        #the axis to lay R values on to

order = np.argsort(self.dates)              #Create an order to sort the dates
dates_s = np.array(self.dates)[order]       #Sorted list of dates
vs = []                                     #V Comparisons sorted
rs = []                                     #R Comparisons sorted

#Assume the number of objects on the first night is the same throughout (should be)
for obj in range(len(self.info[0]))[1:]:
    #Append sorted arrays of the comparison values
    vs.append(np.array(self.info[:,obj,Vresult_index])[order])
    rs.append(np.array(self.info[:,obj,Rresult_index])[order])

#Plot the results. Index is offset from object number by 2, i.e. object 2 is index 0
for i in range(len(vs)):
    #Plot the comparisons in V and R on separate plots
    v_ax.plot(dates_s, vs[i], label = "Obj %d" %(i+2))
    r_ax.plot(dates_s, rs[i], label = "Obj %d" %(i+2))

#Set titles and legends
v_ax.set_title("V Compared to Obj %d" %(self.comparison_index + 1))
r_ax.set_title("R Compared to Obj %d" %(self.comparison_index + 1))
v_ax.legend(loc="upper right")
r_ax.legend(loc="upper right")
```

```
        #Show the plots to the user
        if show:
            figV.show()
            figR.show()

        if saveV:
            figV.savefig("Plots/Bl%d_StandardsV.png" %self.ID)
        if saveR:
            figR.savefig("Plots/Bl%d_StandardsR.png" %self.ID)

    #Shows a plot of the raw instrumental magnitude of the blazar
    #save being set to true will save a copy of the plot and px is the dpi size of the plot
    #If you set show to false, it will not show the plot, useful for saving only
    def ShowRaw(self, show=True, save=False, px=150):
        """
        ShowRaw()
        =========

        Shows the raw, uncalibrated data. Good for a basic idea of the shape of the graph.
        ShowRaw() should look like ShowCalibrated() for nights with good stars.

        Optional Variables:

            show:
                Default = True
                Set to False to suppress displaying the graph

            save:
                Default = False
                Set to True to save a copy of the graph

            px:
                Default = 150
                Alter to adjust the size of the graph
                Used for the dpi optional variable on pyplot

        """
```

```
    fig = plt.figure(dpi=px)          #figure to add data to
    ax = fig.add_subplot(111)         #the axis to lay values on to
    ax.set_title("Bl%d Instrumental Magnitude" %(self.ID))              #Set the title

    order = np.argsort(self.dates)                                      #Create an ordering list
    dates_s = np.array(self.dates)[order]                               #Create an ordered dates array
    vs = np.array(self.info[:,0,Vinstr_index])[order]                   #Ordered instrumental V magnitude array
    rs = np.array(self.info[:,0,Rinstr_index])[order]                   #Ordered instrumental R magnitude array

    ax.plot(dates_s, vs, "-go", label="V")                              #Plot V instrumental magnitude
    ax.plot(dates_s, rs, "-ro", label="R")                              #Plot R instrumental magnitude

    plt.gca().invert_yaxis()                                            #Invert the y axis
    plt.legend(loc="upper right")                                       #Add the legend
    if show:
        plt.show()                                                          #Shot the user

    #Save the file to be used later
    if save:
        fig.savefig("Plots/Bl%d_InstrumentalMagnitude.png" %(self.ID))

#Shows a plot of standardized blazar magnitudes
#Save being set to true will save a copy of the plot and px is the dpi size of the plot
#If you set show to false, it will not show the plot, useful for saving only
def ShowCalibrated(self, show=True, save=False, px=150):
    """
    ShowCalibrated()
    ================

    Displays a graph of the calibrated blazar magnitude in V and in R. This is the final,
    polished result.

    Optional Variables:

        show:
            Default = True
            Set to False to suppress displaying the graph
```

```
    save:
        Default = False
        Set to True to save a copy of the graph

    px:
        Default = 150
        Alter to adjust the size of the graph
        Used for the dpi optional variable on pyplot

"""

fig = plt.figure(dpi=px)        #figure to add data to
ax = fig.add_subplot(111)       #the axis to lay values on to
ax.set_title("Bl%d␣True␣Magnitude" %(self.ID))                    #Set the title

order = np.argsort(self.dates)                      #Create an ordering list
dates_s = np.array(self.dates)[order]               #Create an ordered dates array
vs = np.array(self.info[:,0,Vcal_index])[order]     #Ordered instrumental V magnitude array
rs = np.array(self.info[:,0,Rcal_index])[order]     #Ordered instrumental R magnitude array

#Get the std dev for magnitudes each night for the error bars
vstd = []
rstd = []

#Gather the standard deviations among the observations within each night
#Use std dev in instrumental magnitude
for night in dates_s:
    #Use the std dev in instrumental magnitudes
    vmag = db.project(db.select(db.select(self.data,[(date_col,night),(obj_col,1), \
                        (filter_col,"V")]),[(mag_col,"Flux<0")],mirror=True),[instr_col])
    #Use the std dev in instrumental magnitudes
    rmag = db.project(db.select(db.select(self.data,[(date_col,night),(obj_col,1), \
                        (filter_col,"R")]),[(mag_col,"Flux<0")],mirror=True),[instr_col])
    vstd.append(np.nanstd(vmag))            #Append to the list for the error bars
    rstd.append(np.nanstd(rmag))

ax.plot(dates_s, vs, "go", label="V")               #Plot V instrumental magnitude
ax.plot(dates_s, rs, "ro", label="R")               #Plot R instrumental magnitude
```

```
        ax.errorbar(dates_s, vs, yerr=vstd, capsize=3, fmt="none")          #Add error bars to points
        ax.errorbar(dates_s, rs, yerr=rstd, capsize=3, fmt="none")

        plt.gca().invert_yaxis()                                    #Invert the y axis
        plt.legend(loc="upper right")                               #Add the legend
        if show:
            plt.show()                                                  #Show the user

        #Save the file to be used later
        if save:
            fig.savefig("Plots/Bl%d_Calibrated.png" %(self.ID))

##########################################################################################################
#ERROR FUNCTIONS##########################################################################################
##########################################################################################################

    #Prints out the different types of errors the Blazar has
    def ErrorSummary(self):
        print("Errors:")
        #If there are no errors
        if not 1 in self.e_list:
            print("No Errors For Bl%d" %(self.ID))
        else:
            if self.e_list[0] == 1:
                print("- Object Mismatch: Too Few Objects On:")
                print(str(self.deficient_obj))
            if self.e_list[1] == 1:
                print("- Faint Blazar Warning: Net Counts Zero or Below on Blazar on:")
                print(str(self.blError))
            if self.e_list[2] == 1:
                temp_nc = []            #Display list to show self.nc_obj object (add 1 to all)
                for el in self.nc_obj:
                    temp_nc.append(el+1)
                print("- Net Count Warning: Net Counts for Stars Zero or Below on:")
                print(str(self.ncError))
                print("For Objects " + str(temp_nc))
            elif self.e_list[2] == 2:
                print("- Net Count Warning Resolved: Bad Net Count Stars Discarded")
```

```
            if self.e_list[3] == 1:
                print("- Poor Standard Warning: More Than Threshold Number of Stars Flagged to \
                    be Discarded")
            if self.e_list[4] == 1:
                print("- Poor Solution Warning: More Than Threshold Number of Solutions Flagged to \
                    be Discarded")


    #Finds a mismatched number of objects
    #Assumed the most common object count is the number of objects chosen
    #deletes objects over that number, warns about any under
    def ObjectMismatch(self):

        obj_count = {}      #Dictionary of object number to how many nights have that many objects

        #Go through each night and find the max number of objects that night
        for night in self.dates:
            #Max number of objects that night
            count = int(max(db.project(db.select(self.data,[(date_col,night)]),[obj_col]))[0])

            #Increment max object counts
            if count in obj_count:
                obj_count[count] += 1
            else:
                obj_count[count] = 1

        #Find the key mapped to the highest number, i.e. the most frequent max object
        self.objects = max(obj_count, key=lambda key:obj_count[key])

        #If there are nights with more objects than this, go through and delete them
        if max(obj_count) > self.objects:
            print("Object Mismatch: Excess Object Count")
            print("Assumed Object Count: %d" %(self.objects))
            print("Deleting Excess Objects")

            toss_indices = []           #List of indices to delete

            #Loop through data and prepend the indices so they can be deleted without altering the
```

```
        #next index to delete
        for i in range(len(self.data))[1:]:
            if self.data[i][obj_col] > self.objects:
                toss_indices[:0] = [i]

        for index in toss_indices:
            del self.data[index]

    #If there are nights with too few objects, there isn't an automatic way to handle it
    #Store the nights of issue, alert the user
    if min(obj_count) < self.objects:
        print("Object␣Mismatch:␣Too␣Few␣Object␣Counts")
        print("Flagging␣Deficient␣Nights")

        fewer_nights = []               #Holds the nights with too few objects

        #Go through each night and find those with fewer objects
        for night in self.dates:
            nightly_min = \
                    int(max(db.project(db.select(self.data,[(date_col,night)]),[obj_col]))[0])

            #If the maximun on a certain night is smaller than average object count...
            if nightly_min < self.objects:
                #Note it and alert the user
                fewer_nights.append(night)

        #Alert the user
        print("Too␣few␣objects␣on␣the␣following␣nights:␣" + str(fewer_nights))

        self.deficient_obj = fewer_nights                   #The Blazar will hold this list
        self.e_list[0] = 1                                  #Place the signal

        return -1               #Signal to the program
    else:
        return 0

#Small helper function.
#Takes in a list of lists of one element, and adds their objects to self.nc_obj
```

```
def AddBadObj(self, table):
    for el in table:
        if el[0] not in self.nc_obj:
            self.nc_obj.append(el[0])

#Finds negative or zero net counts
def NCCheck(self):
    #Go through all nights to see if there are any issues with the blazar net counts
    for night in self.dates:
        #Get any object 1 with bad net counts
        hold = db.select(self.data,[(obj_col,1),(mag_col,"Flux<0")])
        if len(hold) > 0:
            #There are bad nights
            self.blError.append(night)          #Add the night to the list of bad blazar net counts
            self.e_list[1] = 1                  #Add the warning flag about a bad blazar



    #Now check for other bad net counts
    #The warning flag in self.e_list will be set if ANY night meets the bad criteria
    #Criteria for bad is as follows:
    #There are 2 or more bad points for the same object in the same filter OR there are three or
    #more bad points in one filter
    #Any less than that and the bad net count will be replaced by the average of the other objects
    #of its kind and "Flux<0" replaced with "Altered"
    for night in self.dates:
        #Each element in Vnets and Rnets will be a list with only the object number inside
        #All V filter net count objects that night
        Vnets = db.project(db.select(self.data,[(date_col,night),(mag_col,"Flux<0"), \
                                                (filter_col,"V")]),[obj_col])
        #All R filter net count objects that night
        Rnets = db.project(db.select(self.data,[(date_col,night),(mag_col,"Flux<0"), \
                                                (filter_col,"R")]),[obj_col])

        #If either one has at least 3 elements, it cannot be automatically handled
        if len(Vnets) >= 3 or len(Rnets) >= 3:
            self.ncError.append(night)
            self.AddBadObj(Vnets)       #Add any new bad objects
```

```
            self.AddBadObj(Rnets)        #Add any new bad objects
        elif len(Vnets) == 2 or len(Rnets) == 2:
            #If one or both has a length of two, it is good unless both are the same object
            if len(Vnets) == 2:
                if Vnets[0] == Vnets[1]:
                    self.ncError.append(night)
            elif len(Rnets) == 2:
                #We only need one or the other, otherwise we will double append the same night
                if Rnets[0] == Rnets[1]:
                    self.ncError.append(night)
            self.AddBadObj(Vnets)        #Add any new bad objects
            self.AddBadObj(Rnets)        #Add any new bad objects
        elif len(Vnets) > 0 and len(Rnets) > 0:
            #If neither criteria was hit, it means the issue can be handled automatically
            print("Fixing␣Minor␣Net␣Count␣Issues␣on␣Bl%d" %(self.ID))
            for el in Vnets:
                #Find the average based on the good points
                Vavg = np.nanmean(db.project(db.select(db.select(self.data,[(obj_col,el[0]), \
                            (filter_col,"V"),(date_col,night)]),[(mag_col,"Flux<0")],mirror=True), \
                            [count_col]))
                #To know which to alter, find the row on that night
                row = db.project(db.select(self.data,[(date_col,night),(obj_col,el[0]), \
                            (filter_col,"V"),(mag_col,"Flux<0")]),[0])[0]
                #Iterate through self.data until you find it
                for i in range(len(self.data)):
                    #Find the row on that date
                    if self.data[i][0] == row and self.data[i][date_col] == night:
                        #Change from "Flux<0" to avoid selecting on it again
                        self.data[i][mag_col] = "Altered"
                        #Replace the bad counts with the average
                        self.data[i][count_col] = Vavg
            for el in Rnets:
                #Repeat all of above for R filter
                Ravg = np.nanmean(db.project(db.select(db.select(self.data,[(obj_col,el[0]), \
                            (filter_col,"R"),(date_col,night)]),[(mag_col,"Flux<0")],mirror=True), \
                            [count_col]))
                #To know which to alter, find the row on that night
                row = db.project(db.select(self.data,[(date_col,night),(obj_col,el[0]), \
```

```
                    (filter_col,"R"),(mag_col,"Flux<0")]),[0])[0]
            #Iterate through self.data until you find it
            for i in range(len(self.data)):
                #Find the row on that date
                if self.data[i][0] == row and self.data[i][date_col] == night:
                    #Change from "Flux<0" to avoid selecting on it again
                    self.data[i][mag_col] = "Altered"
                    #Replace the bad counts with the average
                    self.data[i][count_col] = Ravg

#All bad nights should be noted now
if len(self.ncError) == 0 and len(self.blError) == 0:
    return 0            #No issues
else:
    self.e_list[2] = 1              #Set the error flag
    return -1           #Rephoting might be necessary
```

Bibliography

Cameron Pace, Richard L. Pearson, e. a. 2013, Publications of the Astronomical Society of the Pacific, 125

D'Orazio, D. J., Haiman, Z., & Schiminovich, D. 2015, Nature Journal, 525

Hindman, L. 2018, Bachelor's thesis

Landolt. 2009, Astron.J., 137

M. Villata, e. a. 2002, Memorie della Societa Astronomica Italiana, 73

Raiteri, et al. 2017, Nature, 552

Van Alfen, N., et al. 2018, Research Notes of the AAS, 2, 47

Veron-Cetty, M. P., & Veron, P. 2010, Astron. Astrophys., 518

Index