Using Model Point Spread Functions to Identify Possible Binary Brown Dwarf Systems

Kyle L. Matt

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

Bachelor of Science

Dr. Denise Stephens, Advisor

Department of Physics and Astronomy

Brigham Young University

April 2017

ABSTRACT

Using Model Point Spread Functions to Identify
Possible Binary Brown Dwarf Systems

Kyle L. Matt
Department of Physics and Astronomy, BYU
Bachelor of Science

A Brown Dwarf is a celestial object that forms like a star, but is not massive enough to undergo hydrogen fusion in its core. Due to a lack of an internal energy source, brown dwarf temperatures are not stable with age, making mass and age estimates difficult. Recent evidence suggests that models of brown dwarf evolution are inaccurate; to improve these models, direct mass measurements are required. One of the best methods to empirically measure mass is through observations of the orbital parameters of a binary pair. Angular separation between binary brown dwarfs is often small enough that they are unresolved in images. Due to the great distances from Earth, these brown dwarfs appear as point sources of light and spread out in images in a predictable pattern, due to diffraction, known as the point spread function (PSF). I have developed a Python script based on an older FORTRAN program to find binary systems which are unresolved by creating models of PSFs and testing those models against images taken from the Hubble Space Telescope archive.

Keywords: Brown Dwarf, Binary, HST, PSF, Python

ACKNOWLEDGMENTS

This project was made possible through funding from the BYU Physics and Astronomy Department and a grant from the Office of Research and Creative Activities.

I would like to thank Dr. Denise Stephens for being my advisor, for providing the inspiration for this project and for helping me complete it. I also thank Dr. Mike Joner for being a great boss and mentor during the summers I worked for him at West Mountain Observatory. Dr. Tracianne Neilsen helped proofread this thesis and provided comments that greatly improved it.

I also would like to thank my parents, Edwin and Sharon Matt, for the support they've constantly provided for me and for always encouraging me to try new and hard things. I don't know where I would be without them. My grandfather, Dr. Douglas Henderson, was a great inspiration for me and encouraged my love for the sciences.

Contents

List of Figures

Chapter 1

Introduction

There exists a problem with current models of Brown Dwarf formation and evolution. Dupuy et al. (2014) showed that models underpredicted measured luminosity by a factor of $\approx$2, given mass and age, compared to observations of binary systems HD 130948BC and GI 417BC. They directly measured the masses, ages, and luminosities for these systems and, after comparing with predictions based on models, concluded that the problem likely lies with predicted cooling rates leading to an overestimation of model-derived masses. The current models are poorly constrained. To fully constrain evolutionary models of brown dwarfs empirical and independent measurements of mass, age, and luminosity of more systems must be made. Only a handful of brown dwarfs have had their masses directly measured, due to difficulty in measuring mass. One of the few methods to measure the mass of a celestial object such as a brown dwarf is by inferring the mass from its orbital behavior in a binary system. Thus more binary systems must be discovered.

The purpose of this project is to describe a Python script I wrote to help fit PSF models to data in an effort to detect potential binary systems. In this chapter I describe several concepts that are important to understanding how this script works; including what a brown dwarf is, what a point spread function is, and what cameras are supported. Chapter 2 describes the theory and method used while Chapter 3 describes what each section of the script does. Chapter 4 presents some

preliminary results, my conclusions and necessary future work. Appendix A contains the text of the script along with comments meant to aid the user in understanding it.

1.1 Introduction to Brown Dwarfs

To understand what a brown dwarf is, it is important to understand what a star and planet are. A star is a celestial object comprised of primarily hydrogen and helium that is massive enough to sustain the fusion of hydrogen in its core. This provides an energy source by which the star can produce its own light. Stars form by the gravitational collapse of an interstellar gas cloud and are typically above 80 Jupiter masses (M_J).

A planet, by contrast, is not massive enough to have fusion occur in its core, thus all visible light received from a planet is reflected and only infrared wavelengths are emitted. Planets, as defined by the International Astronomical Union (IAU), orbit a star, are massive enough to be in hydrostatic equilibrium (be spherical), and clear their orbit of similarly sized planetesimals (IAU 2006). They are also expected to form from the accretion disk around a protostar.

A brown dwarf, like a planet, is not massive enough to undergo hydrogen fusion in its core but nevertheless is believed to have formed from the gravitational collapse of interstellar gases, like stars. They are often described as being objects of intermediate mass between the largest planets and the smallest stars, having a mass between 13 and 75 M_J. They emit mainly infrared light with a very small amount of red visible light. These properties make brown dwarfs interesting, as they have characteristics of both stars and planets.

With the exception of possible short lived deuterium fusion early in its life, a brown dwarf has no internal energy source, thus nearly all emitted energy from a brown dwarf originated from gravitational potential when it formed. Cloud structures form in the atmospheres of brown dwarfs and absorb large portions of the electromagnetic radiation spectrum, as shown in figure 1.1, thus

brown dwarf spectra do not match a blackbody curve the way a star might. An object's spectrum is a measurement of electromagnetic intensity by wavelength emanating from that object; what wavelengths dominate a spectrum is primarily determined by the effective temperature, with the majority of light from a brown dwarf being infrared.

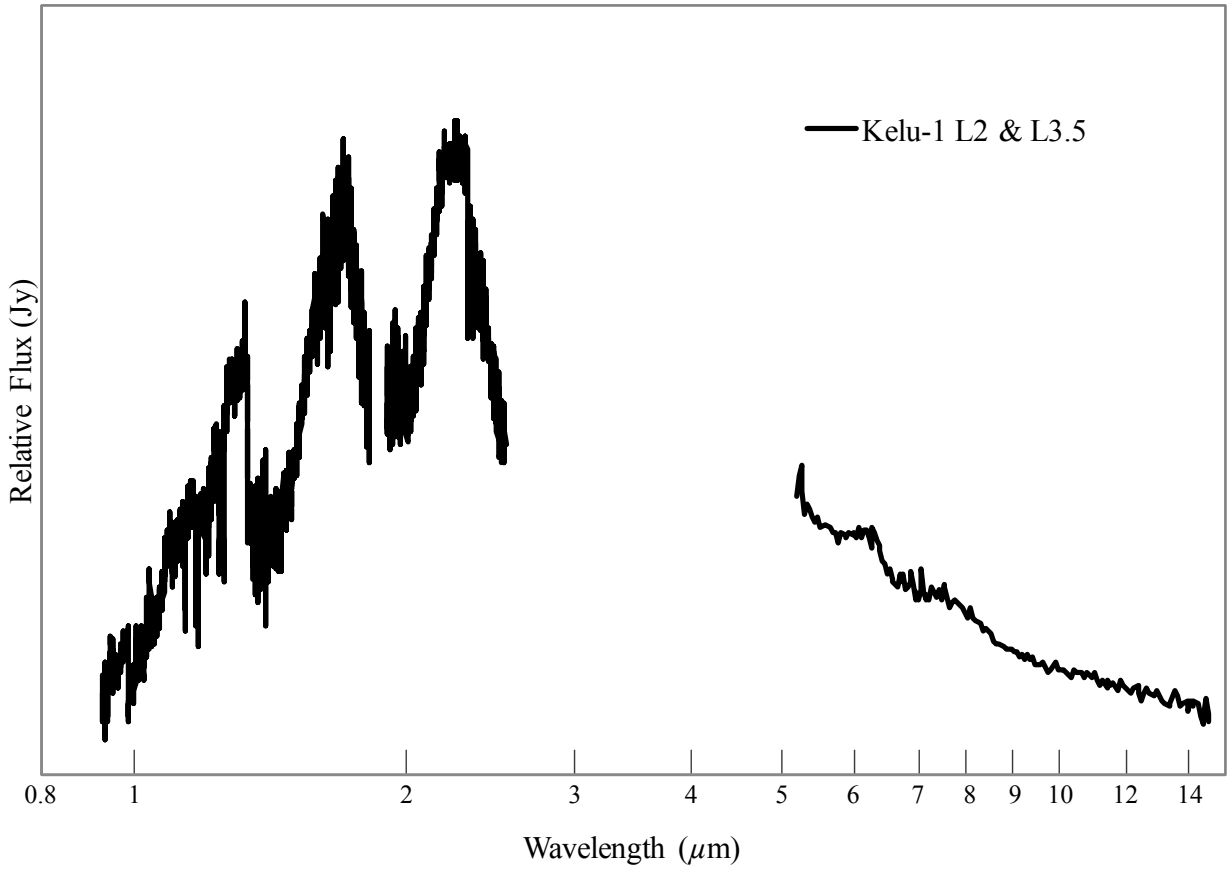

Figure 1.1 *This plot shows a detected spectrum from Kelu-1, constructed using data from the Spitzer Space Telescope archive and Ruiz et al. (1997). Strong absorption features are visible in the jagged shape on the left side. A blackbody curve would be relatively smooth by comparison.*

Brown dwarfs are typically categorized into spectral classifications, like stars, where the temperature largely determines spectral features. For brown dwarfs, the main four spectral types are M-type, L-type, T-type, and Y-type in order from highest to lowest temperature.

1.2 Importance of Binary Systems

Brown dwarfs are difficult to characterize. Since stars undergo fusion in their cores, they have a continuous energy source that is correlated with the star's mass. Thus, stars have a stable mass-luminosity relationship, and their spectra are easily predictable. Brown dwarfs have no such relationship, making it difficult to estimate parameters such as mass, age, and temperature. With a lack of an internal energy source, brown dwarfs cool with time, thus its spectral type can vary over time, requiring a mass-luminosity-age relation to describe (Dupuy et al. 2009). A young, hot, low mass object may have the same spectral characteristics as an old, cool, high mass brown dwarf. Figures 1.2 and 1.3 show roughly how low mass stars (blue), brown dwarfs (green) and large planets (red) evolve with time, taken from an article by Burrows et al. (2001). Should one want to study the formation and evolution of these sub-stellar objects, one would need to determine these among other characteristics.

Mass is the parameter most fundamental to determining how a celestial object will evolve, and in the case of brown dwarfs is one of the most difficult to measure. Isaac Newton's improvement upon Johannes Kepler's third law, shown below, gives a relation between mass (M_1 & M_2), period (P), and semi-major axis (a) with G as the gravitational constant; providing a way to empirically measure masses of objects in a two-body system by observing their orbit. For this reason, identifying binary systems is the best method to measure mass in a brown dwarf system. It is also possible to determine other parameters including age, radius, temperature, and rotation from a binary system (Konopacky 2013).

$$\frac{P^2}{a^3} = \frac{4\pi}{G(M_1 + M_2)}$$

When a binary system has a great enough angular separation it is possible to confirm the existence of a secondary object through simple inspection, as the two objects would appear individually in an image. This is what's known as a visual binary. Figure 1.4 shows an example of a visual

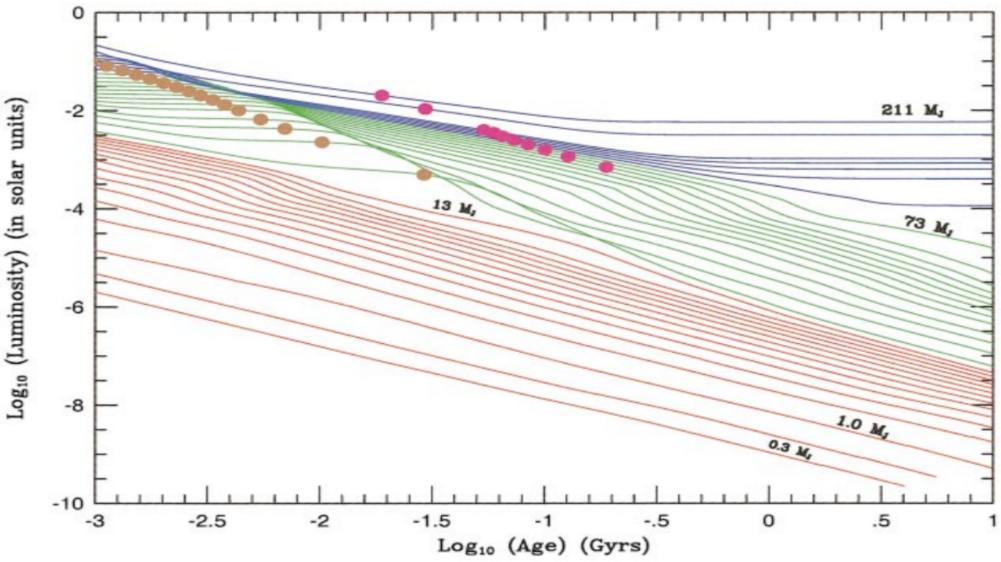

Figure 1.2 *Plot of luminosity vs age of low mass stars, brown dwarfs and planets of several masses (Burrows et al. 2001).*

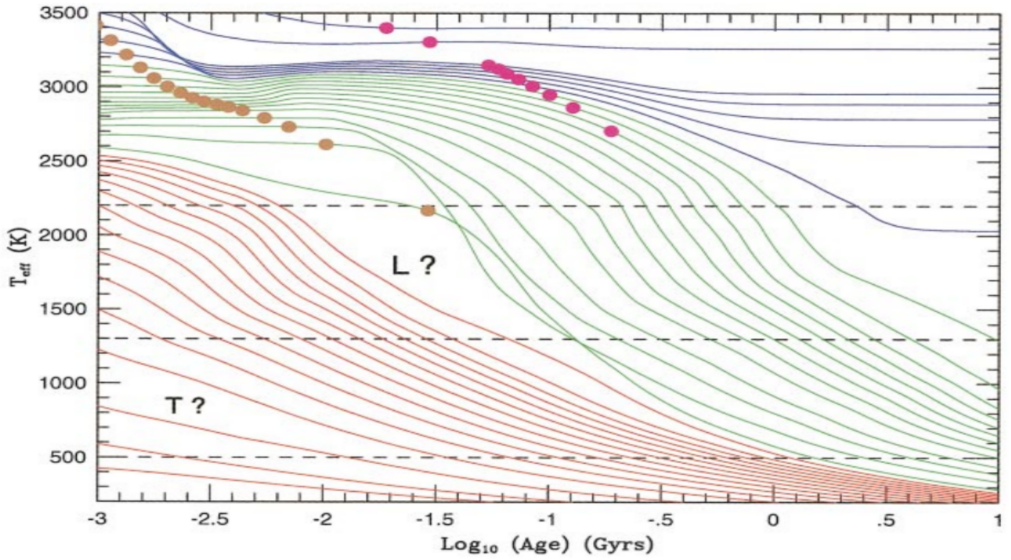

Figure 1.3 *Plot of effective temperatures vs age of low mass stars, brown dwarfs and planets of several masses (Burrows et al. 2001).*

binary, called Kelu-1. When first imaged in 1998, the system appeared as a single object, but in 2005 a subsequent image showed a companion brown dwarf. It turns out their orbit is highly elliptical, thus in 1998 the two objects were close and unresolved while in 2005 they were sufficiently separated to be resolved.

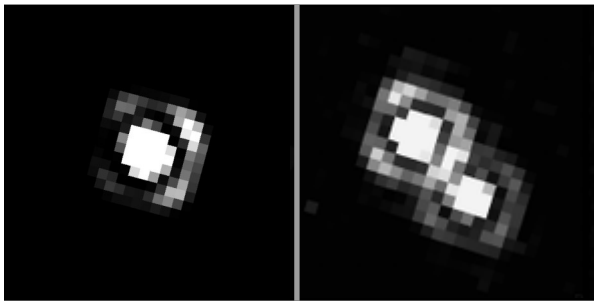

Figure 1.4 *Two images of binary brown dwarf system Kelu-1 taken by the Hubble Space Telescope showing it resolved and unresolved. (NASA et al. 2009)*

The 1998 image should also contain both brown dwarfs, but their separation must be smaller than a pixel. By modeling what an unresolved system would look like, it may be possible to show that a given system is binary even while unresolved.

1.3 The Point Spread Function

A stellar object located many parsecs away acts as a point source of light for an observer on Earth. Therefore, the light from such an object spreads out from a central point in a predictable pattern known as its Point Spread Function (PSF) due to diffraction of light as it enters the telescope. When two such objects have a small angular separation, their respective PSFs overlap to some degree, as seen in Figure 1.5.

When the two objects are closer than the Rayleigh Criterion permits, their PSFs combine to the point that they form a single PSF that cannot be directly resolved into two. The Rayleigh

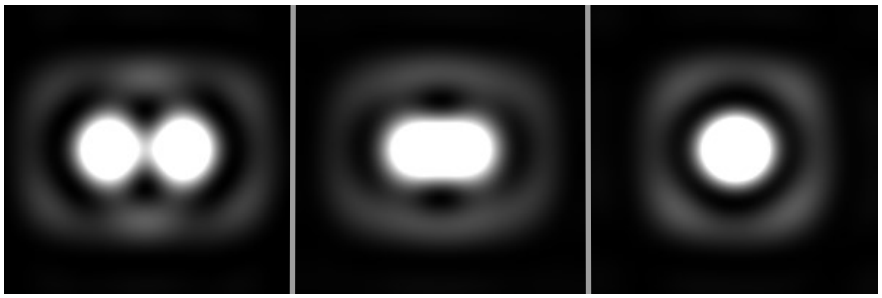

Figure 1.5 *PSFs overlapping to varying degrees.*

Criterion refers to the minimum angular separation two PSFs can have and still be resolved and is approximately equal to $1.22\lambda/D$ where λ is wavelength of light and D is the diameter of the telescope's primary mirror.

Because we know the dimensions of the Hubble Space Telescope (HST) and properties of the cameras and filters, it is possible to model the diffraction and estimate the shape of a PSF. We use software called Tiny Tim (Krist & Hook 2004) to generate model PSFs. Figure 1.6 shows an example of a generated PSF model from Tiny Tim. Compare the PSF in Figure 1.6 with Figure 1.7.

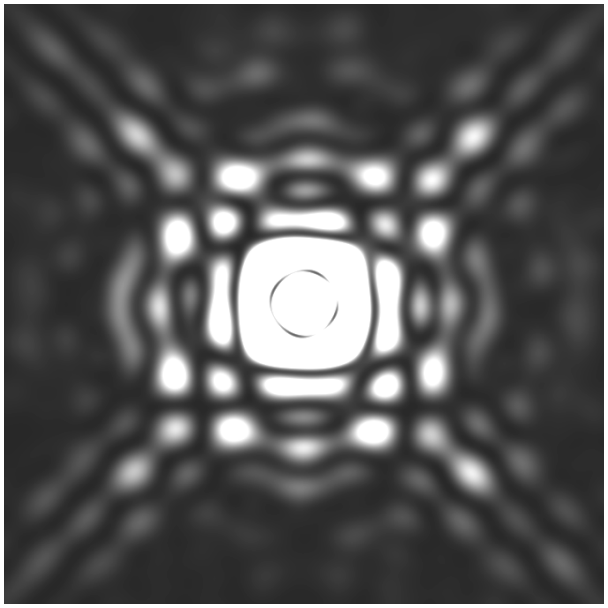

Figure 1.6 *A model point spread function from Tiny Tim.*

1.4 Hubble Instruments and Cameras

The PSF generated by Tiny Tim varies based on a number of factors, including camera, filter used, location of the Brown Dwarf in the image and geometric distortion. HST has a number of instruments and cameras to take data with, for this project we focus on only one of the instruments: the Near Infrared Camera and Multi-Object Spectrometer (NICMOS). Two others will be addressed in a future project, the Advanced Camera for Surveys (ACS), and Wide Field Camera 3 (WFC3). Particular emphasis is be placed on the NICMOS instrument for now because the script described in Capter 3 in its current state is only able to use images from the three cameras of NICMOS.

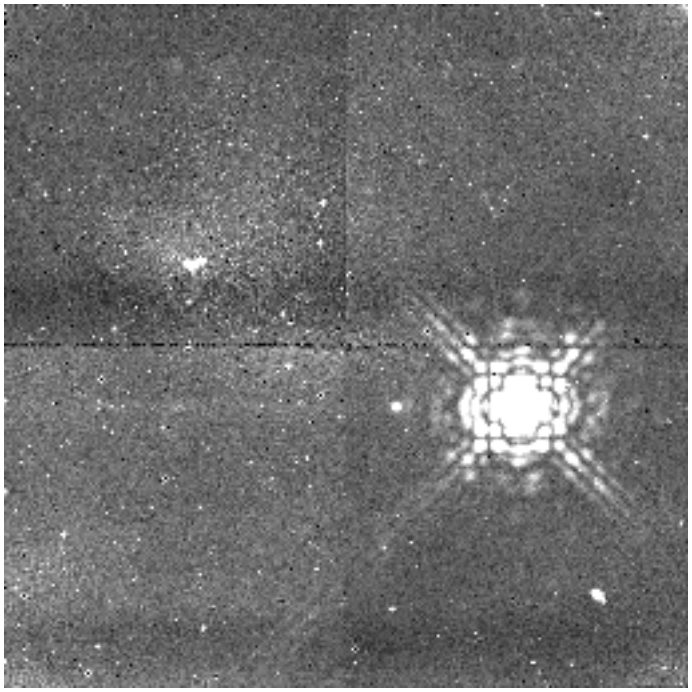

Figure 1.7 *A sample image of a Brown Dwarf system from HST using NICMOS.*

The NICMOS instrument was designed to produce images from near-infrared light, specifically in the range 0.8-2.5 microns dependent upon camera and filter used. NICMOS operated between 1997-1999 and 2002-2008. During its early years, NICMOS was warm, causing thermal emissions

within the telescope itself that exceeded the zodiacal background for wavelengths longer than 1.6 microns (Robberto et al. 2000). This was corrected by the addition of a cryocooler in 2002. Figure 1.7 demonstrates a sample calibrated image from NICMOS depicting a brown dwarf object.

NICMOS has three cameras, each with its own plate scale and set of filters. Camera 1, 2 and 3 have plate scales of about 0.043, 0.075 and 0.20 arcseconds/pixel respectively. Exact values can be found in appendix A.3.

The other two instruments have their own advantages and disadvantages. For example, ACS has a much higher resolution than NICMOS, with plate scales of 0.05 and 0.025 arcseconds/pixel for two of its cameras, but the PSF is significantly distorted. WFC3 is the most advanced instrument in operation on HST as of 2017, thus most future data will be collected by it, until other telescopes are launched such as the Wide Field Infrared Survey Telescope (WFIRST) and the James Webb Space Telescope (JWST). But WFC3 has a lower resolution than NICMOS and ACS.

Future updates will add functionality, enabling the use of ACS and WFC3 data. If feasible, support for WFIRST and JWST instruments may be added as well.

Chapter 2

Method and Procedure

This chapter explains what actions a user should perform in order to prepare data and run the PSF fitting script. The PSF fitting method is described in some detail, but further detail can be found in Chapter 3 and Appendix A, where the code is explained. The method used here is based on the method Stephens & Noll (2006) created to search for binary trans-Neptunian objects.

2.1 Preparing the PSF and Calibrated Image

We use a program called Tiny Tim to generate our model PSFs. It was created by John Krist in 1992 and was designed as an easy to run program to model the PSF for any given instrument, camera and filter of HST (Krist et al. 2011).

- The Tiny Tim program is available for download at: http://tinytim.stsci.edu/sourcecode.php
- The user manual can be found at: http://tinytim.stsci.edu/static/tinytim.pdf
- A web based interface is also available at: http://tinytim.stsci.edu/cgi-bin/tinytimweb.cgi

Before running Tiny Tim, look up which camera, instrument and filter were used (available in

the image header file), the object's coordinates in the image and the approximate spectral type of the brown dwarf. The shape of the PSF depends on all of these parameters.

Each PSF from Tiny Tim is good for one filter and quadrant of the image. Therefore if separate images of the same object use the same filter and are located in the same quadrant of the HST image then the same PSF can be used for both.

The following is a step by step run through of Tiny Tim, bold faced text represents user input.

- To Run Tiny Tim in terminal, type tiny1 followed by the name of the parameter file where tiny1 should save parameters. This can be any desired name, but for this project the convention has been to use the filter name. For this example a filter of F170M is used.

  > tiny1 F170M.in

- Select from the given options which instrument and camera was used, if NICMOS1 was used (after 2002), select option 19.

  : **19**

- Enter the x- and y-coordinates determined earlier, rounded to the nearest integer. For example, if we have coordinates (213.06,104.87) enter

  : **213 105**

- If true, confirm that the image was take after 2002 by entering Y.
  This only applies to NICMOS.

  : **Y**

- Enter the filter name. If F170M were used, enter

  : **f170m**

- Tiny Tim offers several options to build the spectrum. In this project, a user-provided ASCII table of the spectrum was created for various spectral types, this is option 5.

  : **5**

- Give a table with Angstrom and flux pairs as described in Appendix C of the Tiny Tim manual (Krist & Hook 2004).

  : **L4.txt**

- When prompted, select 3 arcsec, or whatever the recommended diameter for the PSF is.

  : **3**

- The PSF must be subsampled by 10 to give us a resolution of 0.1 pixels later on in the PSF fitting script, further explanation is be given in section 3.2.1. Enter

  : **Y**

  : **10**

- Enter 0 for secondary mirror despace

  : **0**

- Finally enter the desired rootname of the PSF image file (with no extension). Our convention is to use the filter name.

  : **F170M**

- This creates a file with parameters to create a PSF, named whatever you chose in the initial step. For NICMOS, run tiny2 with the parameter file to create an undistorted PSF. For ACS and WFC3 run tiny3 to for a PSF that is distorted to match the ACS and WFC3 data.

  > tiny2 F170M.in

  > tiny3 F170M.in

The model PSF are created with the given rootname plus 00. In our example this would be F170M00.fits. Tiny Tim has functionality to create several PSFs at once from a list, so additional generated PSFs would be numbered with 01, 02 ... and so on.

NICMOS data is non-linear, thus preventing accurate magnitude calculations, before we run the PSF fitting script, we need to correct for the non-linearity. This is performed in PyRAF, a python based version of IRAF (Science Software Branch at STScI 2012). IRAF stands for Image Reduction and Analysis Facility and is a collection of software developed by the National Optical Astonomy Observatory (NOAO) for astronomical data reduction and analysis (Tody 2000). PyRAF is a product of the Space Telescope Science Initiative, which is operated by AURA for NASA.
In PyRAF open:

- -> **stsdas**

- -> **hst_calib**

- -> **nicmos**

Now run pedsky on the _cal.fits image and output to a _ped.fits with the same rootname. This step estimates the sky background and quandrant-dependent residual bias (called pedestal) and remove that from the input image.

- -> **pedsky n4mh20jiq_cal.fits n4mh20jiq_ped.fits**

Followed by rnlincor with _ped.fits as input and _rnl.fits as output. This script does the correction for the non-linearlity in the count rate.

- -> **rnlincor n4mh20jiq_ped.fits n4mh20jiq_rnl.fits**

2.2 PSF Fitting

To run the PSF fitting script, place the PSF generated by Tiny Tim and the _rnl.fits files into the same directory along with the scripts *PSFfit.py*, *Functions.py* and *Cameras.json*. Call the

script with the PSF and _rnl.fits file names, ensuring to use Python 3.3 or later with the packages described in section A.1.

> python3 PSFfit.py F170M00.fits n4mh20jiq_rnl.fits

The script prompts the user for x- and y- coordinates, these are the same as those used in the Tiny Tim program, the center coordinates of the object rounded to the nearest whole number.

Due to the stability of PSFs from HST, it is possible to use the model from Tiny Tim to identify a binary system. When two objects are present, their respective PSFs can be superimposed and added together to get a combined binary PSF. The script creates numerous combined PSFs and compares them to the original data to find one that fits the best.

The script creates 100 models of a single PSF, based on the Tiny Tim model, letting the center position vary in 0.1 pixel increments in the x and y directions. This effectively creates a 10x10 grid within the center pixel to search, enabling it to find separations on the order of 0.1 pixels.

First, each of the 100 single PSF models are tested against the data by subtracting the model from the data and measuring the chi square value (χ^2). The model that produces the smallest χ^2 value is selected as the best fit.

Next, binary models are produced by combining two single models while letting position and relative brightness vary. The script iteratively searches for the best binary fit by first letting position vary until a best position is found then refining the relative brightness, repeating these two steps until no major changes occur between iterations. Like for the single fit, each model is tested by finding a χ^2 value and minimizing it.

To determine whether these results indicate the detection of a binary or not, we want to see that the binary fit is significantly better than the single fit. If the best binary fit is more than 3σ better than the best single fit, as measured by $\Delta\chi^2$, we consider the system to be a probable binary.

Figure 2.1 shows a visual representation of what the script does mathematically. Based on the residuals seen, this is likely an image of a binary system.

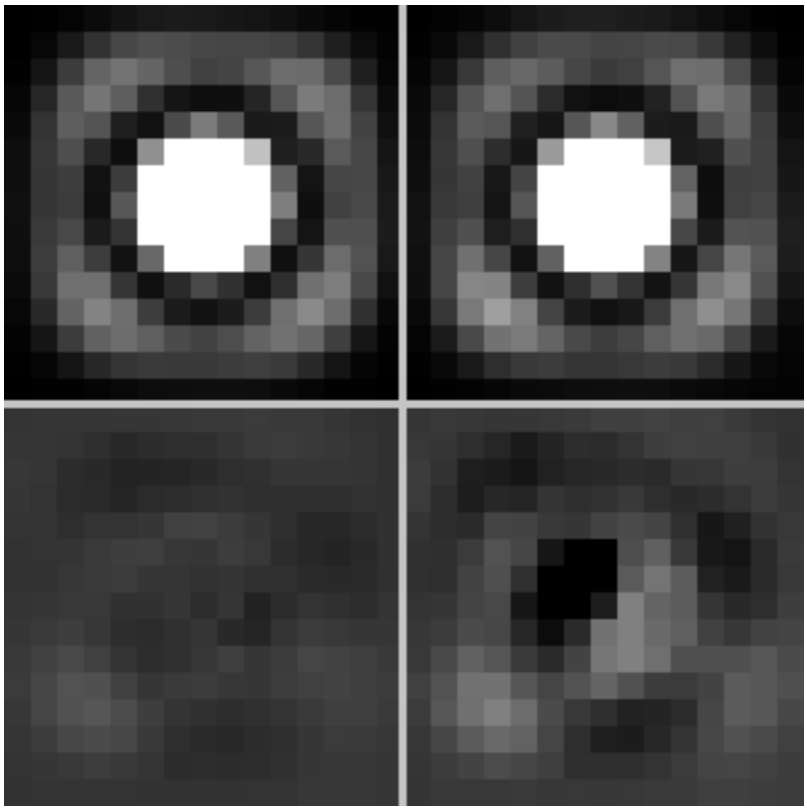

Figure 2.1 *Top left: Data image from HST*
Top Right: Best binary model PSF
Bottom Left: Residuals for binary fit
Bottom Right: Residuals for single fit

2.3 Error

There will always be a level of uncertainty to these calculations. Stephens & Noll (2006) previously produced a program, written in FORTRAN, to use the PSF fitting method, this was later tested by Gardner et al. (2015) to find the probability of detection. Figures 2.2 - 2.4 show his results for the F090M filter of NICMOS 1. For these plots, thousands of fake data were created and the script was run to test the limits of the method by measuring the proportion of brown dwarf binaries that were successfully identified for a given separation and delta magnitude. A similar test will need to be performed on this new script.

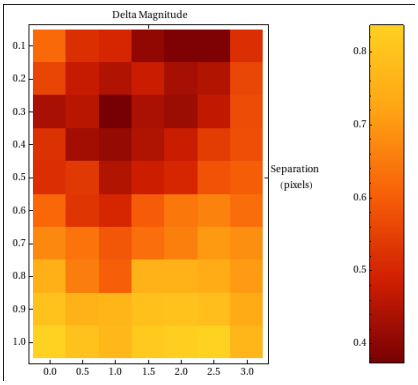

Figure 2.2 *Probability of Detection NICMOS1 F090M Relative Flux*

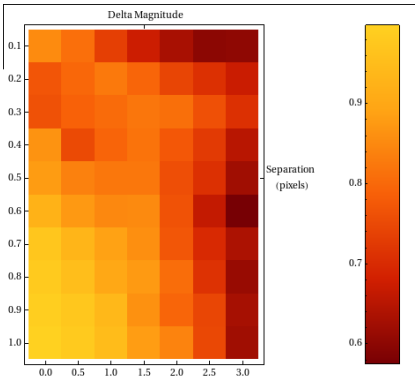

Figure 2.3 *Probability of Detection NICMOS1 F090M Separation*

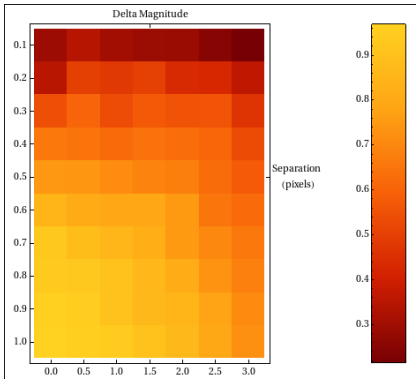

Figure 2.4 *Probability of Detection NICMOS1 F090M Position Angle*

Figure 2.2 describes the proportion of times the FORTRAN program correctly identified the relative brightness of a binary system for various delta magnitudes and separations. Figure 2.3 shows the proportion for correctly detecting the separation and Figure 2.4 for the Position Angle.

Interpreting Figures 2.3 and 2.4 is strait forward, when a binary system has a small separation or if one component is significantly dimmer than the other, it is harder to detect. Figure 2.2 does not appear to follow the same pattern as the other two, the reason for this is not currently known but there is still a bias toward finding objects with greater separation.

It remains to be seen whether this same pattern persists for the Python script, though it is expected to.

Chapter 3

PSF Fitting Script

3.1 Justification for converting from FORTRAN to Python

The first implementation of this method was created using FORTRAN by Stephens & Noll (2006) for the purpose of locating binary trans-Neptunian objects. This script ran fast and worked, but was not very user friendly and a lot of the user's time was spent editing parameter files manually. Handling the data we use has proved to be difficult with FORTRAN. I discussed it with my advisor, Dr. Stephens, and we decided to try writing a new script based on the old one but using Python instead. Our goal here is to reduce the user interaction, to create a single script that is versatile enough to be able to read data from different cameras and instruments and be able to easily add additional camera parameters.

3.2 Code Explained

I will be explaining the script during this section. The script has three components; *PSFfit.py* is the main script that runs the PSF fitting, *Functions.py* is a separate file that only contains functions that *PSFfit.py* uses, at run time a module should be created automatically, finally *Cameras.json* is

a JSON file that contains parameters with unique values from each camera currently supported.

It may be helpful to reference the Appendix while reading this section. Comments precede each portion that I believe requires explanation, commented lines begin with a #. This section only includes the most important parts, or parts that can use more detail about what the script is doing, everything else should be sufficiently explained in the comments.

3.2.1 PSFfit.py

I will break this script up into smaller parts that are easier to explain and lead each explanation with the relevant range of lines as they appear in Appendix A.

18 - 25

Import all required packages:

- *sys* - this package is needed to access input from command-line.
- *math* - a built in mathematical package that enables functions like logarithms, sine and cosine, etc.
- *json* - camera parameters are stored as a json, this package is needed to import them.
- *numpy* - a powerful c compiled numerical package that can be used to perform fast and accurate data manipulation (van der Walt et al. 2011).
- *astropy* - a package specifically designed for use by astronomers, needed to read .fits files.
- *photutils* - a companion package to astropy, the script uses it to perform aperture photometry and fit centroids.

We import all packages that are necessary for this script. Make sure that the version of python used to run this script has at least these packages. The *sys*, *math*, and *json* packages come built in with Python, the rest must be downloaded separately. The comments on lines 12-17 describe which versions are required.

47 - 54

When we generated the PSF with Tiny Tim, we subsampled by 10. This means that the image we got out was 10 times as wide and 10 times as long as a PSF would appear in an HST image. This portion of the script takes the over-sized model PSF and bins it back down to the same dimensions as the data, but allowing the center to vary in 1 pixel increments before doing so. This allows us to vary the position of the center by 0.1 pixel increments at the resolution of the HST image. Now we can treat the center pixel of the PSF as having a 10x10 grid; the script generates 100 different single PSF models, one for each coordinate on that grid.

'*f.binsetion*' is a function from *Functions.py* (as are all functions beginning with f. in this script), how it works is described in Appendix A.2. It takes a bigger array and bins by the values given, here binning the PSF by 10.

These arrays are stored on a table until all 100 are made then they are transfered to an array. Thus array_of_PSFs is a 3d array where each slice is an array with a PSF image. The id of each array (00 - 99) is two digits where each digit corresponds to the displacement of that pixel in the y and x directions.

101 - 126

This while loop calculates the best single fit. It does so with the help of the f.minchi and f.fitpsf1 functions described in Appendix A.2.

f.minchi takes the block of model PSFs and iterate through them, subtracting each from the real data until a model is found that produces the lowest χ^2 value.

f.fitpsf1 takes the single PSF that fit best and try out numerous multiples of the base value until it finds a flux that minimizes χ^2.

Since both *f.minchi* and *f.fitpsf1* use a parameter the other produces, each are repeated until a stable best single fit solution is met. If no best single fit is found and the functions never stabilize, the while loop is set to end after 10 loops to prevent an infinite loop.

152 - 208

Nine pixels are selected as potential locations of the secondary, with the primary assumed to be the central pixel, the nine pixels are the center pixel and the eight surrounding pixels. For now this is sufficient to show proof of concept, future optimization will enable a larger search radius and smarter pixel selection.

This section is the binary fit algorithm. The script iterates through each potential secondary coordinate. Since the strength of the signal drops off quickly, we only care about the very center of the PSF.

f.binarybase takes the center of the primary on a 5x5 grid (2,2), and the center of the secondary (the nine central pixels) and return three 5x5 arrays, w1 with all zeros except the nine surrounding the point (2,2), w2 is the same but around the secondary's coordinate, w is a combination of both. These are used to only consider the pixels immediately around the expected centers by setting all other pixels to zero.

The while loop (165-172) works similarly to the single fit function earlier, iterating until values stabilize.

f.twopsf calculates the best primary and secondary coordinates that minimize χ^2. This is only an approximate though.

f.tworef takes the best primary and secondary coordinates and test just those and the surrounding eight pixels, allowing the relative flux to vary more finely. It gives back the primary, secondary and relative flux between them that minimizes χ^2.

f.fitpsf2 finds the total flux that fits best.

The next while loop (183-192) refines the flux of the primary and secondary.

f.scpsf2 will initialize a value for the secondary flux before the while loop.

f.rfpsf is used in the loop, alternating with primary and secondary as input, to refine the relative flux.

210 -222

The binary fit algorithm produces a list of tuples, where each element of the list corresponds to a potential secondary coordinate as described above. Only elements 0-8 of each tuple are defined in the binary fit algorithm, several others are added later. Here I list all output elements, both added here and later. The elements of each tuple are as follows.

- output[0] : Sum of residual errors
- output[1] : Coordinates of the primary
- output[2] : Coordinates of the secondary
- output[3] : ID of the PSF that best fits the primary
- output[4] : Flux of the primary
- output[5] : ID of the PSF that best fits the secondary
- output[6] : Flux of the secondary
- output[7] : Total flux of the system
- output[8] : χ^2 value of the model
- output[9] : Position angle between primary and secondary

- output[10]: Separation in arcseconds
- output[11]: Magnitude of primary
- output[12]: Magnitude of secondary

240 - 262

The best models of the primary and secondary are here used to calculate a center coordinate to a greater accuracy than 1 pixel. The centroid function from *photutils* is used to find the center of each PSF, these values replace the integer coordinates determined in the best binary fit algorithm.

The new center coordinates are used to calculate the angular separation between the primary and secondary and their position angle. The center is calculated using Pythagorean theorem while the angle is determined with an arctangent added to the orientation angle HST had when the image was taken.

274 - 297

Finally, the script will turn the flux calculated earlier into magnitudes. The *photutils CircularAperture* and *aperture_photometry* packages are useful for calculating the number of counts in a circular aperture around the central position. The radius of the aperture depends on the camera and is extracted from the header file. The *FNU* value extracted from the header file is used to convert from counts to flux in units of *Janskys*.

$F\nu$ is the flux from the star Vega, using the same filter, and allows us to convert flux into magnitudes using Vega as a zero point. The *apcorr* value is a correction to the aperture value. Both of these values are read in from the *Cameras.json* file.

Magnitude calculated as

$$M = -2.5\ log_{10}(\frac{FNU * counts * apcorr}{F\nu})$$

3.2.2 Functions.py and Cameras.json

Functions.py contains several custom functions that I wrote to assist *PSFfit.py* run. These were originally located in the body of *PSFfit.py*, but I found it convenient to move them into their own file for several reasons.

First, the script as it exists now is slow, taking around seven to ten minutes per image when it ideally would run in a minute or less. A future goal is to expand the region that *PSFfits.py* searches during the binary fit, but doing so will increase run time. To speed the script up, several of the functions can be converted to C code, or Cython code, as compiled code runs faster than scripts. By extracting the functions from *PSFfit.py* it should be easier to make this conversion later.

I also found that having the functions separate made the main script much more readable for a user. Since other people besides me will likely use this code, I opted to clean up by moving the functions.

An explanation of what each function does can be found in Appendix A.2.

This script is meant to be versatile enough to work with different cameras, and eventually different instruments. To do this, I created a camera parameter file called *Cameras.json*. I chose to use a JSON file type because Python has a built in package able to handle JSON files, importing the data directly into a Python dictionary data structure.

Eventually cameras from other instruments will be able to be added to *Cameras.json* to enable their use. Some editing of *PSFfit.py* and *Functions.py* will have to occur before ACS, WFC3 or other instruments can be added however.

Chapter 4

Results and Conclusions

4.1 Results and Analysis

To test the script, I ran it with input from three different brown dwarf systems. One, Kelu-1, is a known binary system while the other two, 2MASS0559-14 and 2MASS0036+18, are systems that we suspect may be binary but are not currently known to be. The following tables summarize my preliminary results. Note, we are not currently ready to convert the $\Delta\chi^2$ value into a p-value or σ. For now I will be presenting results in units of $\Delta\chi^2$.

Kelu-1 Results

Date (UT)	Camera	Filter	Separation (arcsecs)	Position Angle (degrees)	Primary Magnitude	Secondary Magnitude	$\Delta\chi^2$
1998-06-19	NIC3	F187N	0.1614	50.93	13.189	13.322	4.3205
1998-06-19	NIC3	F187N	0.1643	51.00	12.969	13.332	3.8295
1998-06-19	NIC3	F187N	0.2072	33.78	12.472	14.875	3.3881
1998-08-14	NIC1	F165M	0.0457	33.98	12.988	13.171	9.4253
1998-08-14	NIC1	F145M	0.0503	55.18	13.773	14.180	20.688

2MASS0559-14 Results

Date (UT)	Camera	Filter	Separation (arcsecs)	Position Angle (degrees)	Primary Magnitude	Secondary Magnitude	$\Delta\chi^2$
2004-10-20	NIC1	F090M	0.0302	144.5	17.121	17.797	0.2471
2004-10-20	NIC1	F110M	0.0439	141.5	14.726	17.623	0.3074
2004-10-20	NIC2	F222M	0.0375	62.0	14.689	14.709	0.1317
2004-10-20	NIC2	F222M	0.0505	35.4	14.033	16.961	0.1340

2MASS0036+18 Results

Date (UT)	Camera	Filter	Separation (arcsecs)	Position Angle (degrees)	Primary Magnitude	Secondary Magnitude	$\Delta\chi^2$
2005-01-03	NIC1	F110W	0.0182	156.9	13.846	13.906	5.3299
2006-10-10	NIC1	F190N	0.0336	53.5	12.474	12.571	0.0295
2006-10-10	NIC1	F164N	0.0336	53.5	12.045	12.642	0.0667
2006-10-10	NIC1	F090M	0.0330	70.9	14.938	16.134	1.8005
2006-10-10	NIC1	F145M	0.0423	69.4	12.557	15.121	0.6146
2006-10-10	NIC1	F160W	0.0327	70.1	12.235	13.248	0.7420

The results for Kelu-1 are promising; the data from the 19th of June 1998 all have roughly the same separation and two of them have the same position angle. Though the results are not perfect, especially when considering the August 14th data where the separation is significantly smaller. This may be due to a number of reasons; the secondary may be too faint in the filter used or the separation may be too small. It is also possible the script is not currently using the optimal method for PSF fitting. We will need to analyze the data more and decide if the script needs improvement or not.

The results from 2MASS0559-14 do not appear to be very conclusive. The separations, position angles and magnitudes are not consistent across all images tested and the $\delta\chi^2$ values are very low. I do not see a strong signal indicating a binary system. It is important to remember that not

finding evidence of a binary system does not mean the system is definitely single, just that if a companion exists it was not successfully detected.

2MASS0036+18 shows some promise, the final three entries in the table have a rather consistent separation and position angle, however the magnitudes are not consistent. This is an object that will be worth observing in the future.

4.2 Conclusions and Future Work

These preliminary results suggest that the PSF fitting script works, it successfully found the Kelu-1 binary and to a lesser extent suggests 2MASS0559-14 and 2MASS0036+18 have potential to be binary. The problem is that the results are not always consistent.

The script currently uses χ^2 to test for best fit, though it is not immediately obvious that this is the best method. We will want to try other measures, including the mean square error and absolute value of residual errors, to see which is best at identifying the best fit. Resolving this issue should be the first priority for future work in this project. We will then need to run an error analysis on this new script.

Another problem that needs to be addressed is the speed issue. In its current form, the script takes too long to run, taking around ten minutes per image. The script can be sped up by rewriting several of the functions in a compiled language such as C.

A large amount of data exists that was collected by the ACS and WFC3 instruments, this script currently cannot work with data from them however. The script will need to be updated in the future to be able to use other HST instruments and cameras.

Bibliography

Astropy Collaboration, Robitaille, T. P., Tollerud, E. J., et al. 2013, A&A, 558, A33

Bradley, L., Sipocz, B., Robitaille, T., et al. 2016, astropy/photutils: v0.3, doi:10.5281/zenodo.164986

Burrows, A., Hubbard, W. B., Lunine, J. I., & Liebert, J. 2001, Rev. Mod. Phys., 73, 719

Dupuy, T. J., Liu, M. C., & Ireland, M. J. 2009, The Astrophysical Journal, 692, 729

—. 2014, The Astrophysical Journal, 790, 133

Gardner, D., Stephens, T., Stephens, D., & Salway, E. 2015, in American Astronomical Society Meeting Abstracts, Vol. 225, American Astronomical Society Meeting Abstracts, 138.41

IAU. 2006, IAU Resolution B5: Definition of a Planet in the Solar System, Available at: https://www.iau.org/static/resolutions/Resolution_GA26-5-6.pdf

Konopacky, Q. M. 2013, Mem. S. A. It., 84, 1005

Krist, J., & Hook, R. 2004, The Tiny Tim User's Guide, ver 6.3, Available at: http://tinytim.stsci.edu/static/tinytim.pdf

Krist, J. E., Hook, R. N., & Stoehr, F. 2011, in Proc. SPIE, Vol. 8127, Optical Modeling and Performance Predictions V, 81270J

NASA, ESA, & Stumpf, M. 2009, Binary brown dwarf Kelu-1, https://www.spacetelescope.org/images/opo0901a/

Robberto, M., Sivaramakrishnan, A., Bacinski, J. J., et al. 2000, in , Vol. 4013, 386–393

Ruiz, M. T., Leggett, S. K., & Allard, F. 1997, The Astrophysical Journal Letters, 491, L107

Science Software Branch at STScI. 2012, PyRAF: Python alternative for IRAF, Astrophysics Source Code Library, ascl:1207.011

Stephens, D. C., & Noll, K. S. 2006, The Astronomical Journal, 131, 1142

Tody, D. 2000, Bristol: Institute of Physics Publishing, doi:10.1888/0333750888/2923

van der Walt, S., Colbert, S. C., & Varoquaux, G. 2011, Computing in Science & Engineering, 13, 22

Appendix A

Code

A.1 PSFfit.py

```
1  # Written by Kyle L. Matt
2  # Brigham Young University
3  # April 2017
4  #
5  # Make sure dependencies are met and ensure that PSFfit.py,
6  # Functions.py, and Cameras.json are in the same directory.
7  #
8  # enter to run:
9  # python PSFfits.py [PSF_rootname].fits [Hubble_data_rootname].fits
10 #
11 # Dependencies:
12 # python 3.3 or later
13 # numpy 1.8 or later
14 # astropy 1.0 or later
15 # photutils 0.3.2 or later
16 # sys, math, and json are part of the standard library in python and should
```

```
17 # be included in your version.
18 import sys
19 import numpy as np
20 import math
21 import json
22 from astropy.io import fits
23 from photutils import centroid_2dg
24 from photutils import CircularAperture
25 from photutils import aperture_photometry as phot
26
27 # Functions refers to the associated Functions.py
28 import Functions as f
29
30 # Load in camera data and then access PSF image, storing its data in a numpy
31 # array 'Fitsdata'
32 # sys.argv[1] accesses the input from command-line following the script name
33 # Fitsdata stores the pixel array from the PSF.
34 cameras=json.loads(open('Cameras.json').read())
35 infile = sys.argv[1]
36 print('- - -\nBegin binning of ' + infile)
37 FitsFile = fits.open(infile)
38 if len(FitsFile) != 1:
39     frame = 'SCI'
40 else:
41     frame = 0
42 Fitsdata = FitsFile[frame].data
43 FitsFile.close()
44
45 # Shift the PSF by 1 pixel in both x and y directions, bin the array by 10
46 # after each shift, storing the resulting arrays into array_of_PSFs.
```

```
47 nbin = 10
48 ymax,xmax = Fitsdata.shape
49 psf_table=[]
50 for i in range(1,nbin+1):
51     for j in range(1,nbin+1):
52         temp_array=Fitsdata[j:ymax-(nbin-j),i:xmax-(nbin-i)]
53         psf_table.append(f.binsection(temp_array,nbin,nbin))
54 array_of_PSFs = np.asarray(psf_table)
55
56 # Access data image, storing its data into 'Fitsdata' replacing the PSF
57 # and store the header info into 'Fitshead'.  Record entry on telescope,
58 # instrument, camera and filter.
59 # sys.argv[2] accesses the second entry from command-line
60 infile = sys.argv[2]
61 FitsFile = fits.open(infile)
62 if len(FitsFile) != 1:
63     frame='SCI'
64 else:
65     frame = 0
66 Fitsdata = FitsFile[frame].data
67 Fitshead = FitsFile[0].header
68 FitsFile.close()
69 telescope = Fitshead['TELESCOP']
70 instrument = Fitshead['INSTRUME']+str(Fitshead['CAMERA'])
71 filtername = Fitshead['FILTER']
72
73 # Create output file.  Initialize it with data file rootname, filter,
74 # and the date and time of observation.
75 print('---\nBegin selection of best single fit')
76 OutFileName = Fitshead['ROOTNAME'] + '_output.txt'
```

```
77  out = open(OutFileName,'w')
78  out.write(Fitshead['ROOTNAME']+' - '+filtername+'\n')
79  out.write('UT '+Fitshead['DATE-OBS']+' '+Fitshead['TIME-OBS']+'\n')
80
81  # Prompt user for x- and y-coordinates, store coordinates as tuple 'center'
82  print('Please enter the coordinates of the center of the object')
83  center=input('in the format "XXX YYY" with a space between each number\n')
84  center=(int(center.split()[1]),int(center.split()[0]))
85  realpsf = Fitsdata[center[0]-3:center[0]+2,center[1]-3:center[1]+2]
86
87  # Initialize several parameters.  Sigma=0 for NIMCOS but may need
88  # to change for other instruments.
89  bkgd = cameras[instrument]['Background']
90  realpsf=realpsf-bkgd
91  realpsf[realpsf<0]=0
92  base=np.sum(realpsf,axis=(0,1))
93  sigma = 0.0
94  realpsf[realpsf<=sigma]=0
95
96  # Here the script begins to search for the best single fit.  It iteratively
97  # tests each of the 100 PSF models against the data, minimizing chi^2.
98  # f.minchi finds a best single fit, testing all 100 models.
99  # f.fitpsf1 uses that model and finds a flux that gives the best fit.
100 # This repeats until flux does not change between iterations (10 times max).
101 flux=base
102 iteration=0
103 comp=0
104 minchi_val=0
105 out.write('Single Fit:\n')
106 weights=[[0,0,0,0,0],
```

```
107             [0,1,1,1,0],
108             [0,1,1,1,0],
109             [0,1,1,1,0],
110             [0,0,0,0,0]]
111 print('iter: %i  psf: ?  flux:%9.4f'%(iteration,flux))
112 while np.round(flux,6)!=np.round(comp,6) and iteration < 10:
113     comp=flux
114     single_fit = f.minchi(array_of_PSFs,realpsf,flux,weights)
115     minchi_val,flux=f.fitpsf1(array_of_PSFs[single_fit],realpsf,base,weights)
116 
117     cx=int((array_of_PSFs[single_fit].shape[1]-1)/2)
118     cy=int((array_of_PSFs[single_fit].shape[0]-1)/2)
119     temp=array_of_PSFs[single_fit][cy-2:cy+3,cx-2:cx+3]*flux
120     iteration+=1
121 
122     print('iter: %i  psf:%2i  flux:%9.4f  minchi:%9.4f'
123         %(iteration,single_fit,flux,minchi_val))
124 out.write('iter: %i  psf:%2i  flux:%9.4f  minchi:%9.4f\n'
125     %(iteration,single_fit,flux,minchi_val))
126 print('---\nBegin selection of best binary fit')
127 
128 # Select the pixels to be tested while searching for the secondary.
129 # For now this is the center pixel and the 8 surrounding pixels.
130 # If any pixels are not located in the array, they are removed.
131 coords=[]
132 coords.append((center[0]-1,center[1]-1))
133 coords.append((center[0]-1,center[1]+0))
134 coords.append((center[0]-1,center[1]+1))
135 coords.append((center[0]+0,center[1]-1))
136 coords.append((center[0]+0,center[1]+0))
```

```
137 coords.append((center[0]+0,center[1]+1))
138 coords.append((center[0]+1,center[1]-1))
139 coords.append((center[0]+1,center[1]+0))
140 coords.append((center[0]+1,center[1]+1))
141 temp=[]
142 for secondary in coords:
143     if not any(i < 0 for i in secondary):
144         if secondary[0] < Fitsdata.shape[0]:
145             if secondary[1] < Fitsdata.shape[1]:
146                 temp.append(secondary)
147 coords=temp
148
149 # Begin the binary fit routine.  This section will, for each set of pixels
150 # defined above, search for a combination of two PSF models that combined
151 # make the binary PSF that best fits the data.
152 bestP,bestS,bestRF=0,0,0
153 outputs=[]
154 for secondary in coords:
155     pcoords=(2,2)
156     scoords=(secondary[0]-center[0]+2,secondary[1]-center[1]+2)
157     w,w1,w2=f.binarybase(pcoords,scoords)
158     iteration=0
159     base = np.sum(w*realpsf)
160     comp=0
161     flux=base*1
162
163     # This loop selects the two PSF models that combine to produce the best
164     # binary fit.  Relative flux is coarsely tested here, refined later.
165     while np.round(flux,6)!=np.round(comp,6) and iteration < 10:
166         comp=flux
```

```
167         bestP,bestS=f.twopsf(array_of_PSFs,scoords,realpsf,flux,w)
168         bestP,bestS,bestRF=f.tworef(array_of_PSFs,scoords,realpsf,
169                                     flux,w,bestP,bestS)
170         flux=f.fitpsf2(array_of_PSFs[bestP],array_of_PSFs[bestS],
171                        scoords,realpsf,base,w,bestRF)
172         iteration+=1
173     bestPrim=array_of_PSFs[bestP]
174     bestSec=array_of_PSFs[bestS]
175
176     compprim=1.0
177     compsec=1.0
178     fluxprim=2.0
179     fluxsec=f.scpsf2(bestPrim,bestSec,scoords,w2,realpsf,flux,bestRF)
180     iteration=0
181
182     # This while loop refines the relative flux of the two PSFs.
183     while np.round(fluxprim,6)!=np.round(compprim,6) and np.round(fluxsec,6)!=
            np.round(compsec,6):
184         compprim = fluxprim
185         compsec = fluxsec
186         if iteration >= 10:
187             break
188         fluxprim=f.rfpsf(bestPrim,pcoords,bestSec,
189                          scoords,w1,realpsf,base,fluxsec)
190         fluxsec=f.rfpsf(bestSec,scoords,bestPrim,
191                         pcoords,w2,realpsf,base,fluxprim)
192         iteration+=1
193     cx=int((bestPrim.shape[1]-1)/2)
194     cy=int((bestPrim.shape[0]-1)/2)
195     psf1=bestPrim[cy-pcoords[0]:cy-pcoords[0]+5,cx-pcoords[1]:cx-pcoords[1]+5]
```

```
196       psf2=bestSec[cy-scoords[0]:cy-scoords[0]+5,cx-scoords[1]:cx-scoords[1]+5]
197
198       # sumpsf is the best binary fit.
199       sumpsf=psf1*fluxprim+psf2*fluxsec
200       resid=np.sum(w*np.abs(realpsf-sumpsf))
201       residerr=resid/np.sum(w)
202       chi_val = np.sum((w*(realpsf-sumpsf)**2)/sumpsf)
203
204    # Output the data collected to command-line and to the outputs tuple
205       print('psf1:%2i   flux1:%9.4f   psf2:%2i   flux2:%9.4f   resid:%.5f'
206             %(bestP,fluxprim,bestS,fluxsec,residerr))
207       outputs.append((residerr,center,secondary,bestP,
208                           fluxprim,bestS,fluxsec,fluxprim+fluxsec,chi_val))
209 # Here are the entries in each entry of outputs, both added here and later:
210 # output[0] : Sum of residual errors
211 # output[1] : Coordinates of the primary
212 # output[2] : Coordinates of the secondary
213 # output[3] : ID of the PSF that best fits the primary
214 # output[4] : Flux of the primary
215 # output[5] : ID of the PSF that best fits the secondary
216 # output[6] : Flux of the secondary
217 # output[7] : Total flux of the system
218 # output[8] : Chi^2 value of the model
219 # output[9] : Position angle between primary and secondary
220 # output[10]: Separation in arcseconds
221 # output[11]: Magnitude of primary
222 # output[12]: Magnitude of secondary
223
224 print('---\nBegin calculation of angles and separation')
225 orient=Fitshead['orientat']%360
```

```
226 bestprim=[]
227 bestsec=[]
228 tempout=[]
229 cx=int((array_of_PSFs[0].shape[1]-1)/2)
230 cy=int((array_of_PSFs[0].shape[0]-1)/2)
231 PlateScaleX=cameras[instrument]['PlateScaleX']
232 PlateScaleY=cameras[instrument]['PlateScaleY']
233
234 # Here, the position angle and separation are calculated.
235 # First, the position of the center of each model PSF is determined
236 # by fitting a Gaussian centroid, these more exact centers replace
237 # the integer coordinates in the outputs list.
238 # The more exact centers are used to find a separation between the
239 # center of the primary and secondary and the position angle.
240 for itr,output in enumerate(outputs):
241
242     xp=output[1][1]+centroid_2dg(array_of_PSFs[output[3]])[0]-cx
243     yp=output[1][0]+centroid_2dg(array_of_PSFs[output[3]])[1]-cy
244     xs=output[2][1]+centroid_2dg(array_of_PSFs[output[5]])[0]-cx
245     ys=output[2][0]+centroid_2dg(array_of_PSFs[output[5]])[1]-cy
246
247     x1=cx+centroid_2dg(array_of_PSFs[output[3]])[0]-cx
248     y1=cy+centroid_2dg(array_of_PSFs[output[3]])[1]-cy
249     x2=cx+centroid_2dg(array_of_PSFs[output[5]])[0]-cx
250     y2=cy+centroid_2dg(array_of_PSFs[output[5]])[1]-cy
251
252     # the angle between the primary and secondary in this image is
253     # converted into an absolute value by adding the orientation angle
254     angle=(orient-90+math.atan2(ys-yp,xs-xp)*180/math.pi)%180
255     if (ys-yp,xs-xp)==(0,0):
```

```
256             angle = 0.0
257         sep=(((xp-xs)*PlateScaleX)**2+((yp-ys)*PlateScaleY)**2)**(1/2)
258         tempout.append((output[0],)+((yp,xp),(ys,xs))+output[3:9]+(angle,sep))
259
260         bestprim.append((array_of_PSFs[output[3]]*output[4],(y1,x1)))
261         bestsec.append((array_of_PSFs[output[5]]*output[6],(y2,x2)))
262 outputs=tempout
263
264 print('---\nBegin calculation of magnitudes')
265
266 FNU=Fitshead['PHOTFNU'] # Converts counts to flux in Janskys
267 Fv=cameras[instrument]['Filters'][filtername]['Fv']
268 apcorr=cameras[instrument]['Filters'][filtername]['apcorr']
269 magsout=[]
270 ap_rad = cameras[instrument]['Aperture'] # Radius of the aperture
271
272 # Calculate the magnitudes of the Primary and Secondary based on
273 # the flux (counts) and a Vega zero point.
274 for prim,sec in zip(bestprim,bestsec):
275     # Primary
276     apertures = CircularAperture(prim[1],r=ap_rad)
277     ap_table=phot(prim[0],apertures)
278     counts=ap_table['aperture_sum'][0]
279     if FNU*counts*apcorr/Fv>0:
280         Pmag=-2.5*math.log(FNU*counts*apcorr/Fv,10)
281     else:
282         Pmag=float('NaN')
283     # Secondary
284     apertures = CircularAperture(sec[1],r=ap_rad)
285     ap_table=phot(sec[0],apertures)
```

```
286         counts=ap_table['aperture_sum'][0]
287         if FNU*counts*apcorr/Fv>0:
288             Smag=-2.5*math.log(FNU*counts*apcorr/Fv,10)
289         else:
290             Smag=float('NaN')
291 
292         magsout.append((Pmag,Smag))
293 
294 # Sort outputs by residuals and write to the output file
295 tempout=[]
296 for output,mags in zip(outputs,magsout):
297     tempout.append(output+mags)
298 outputs=tempout
299 out.write('\t---\nBinary Fit:')
300 outputs.sort(key=lambda x: x[0])
301 for itr,output in enumerate(outputs):
302     out.write('\n'+str(itr+1)+'    error:%.4f     chi^2:%.4f\n'
303         %(output[0],output[8]))
304     out.write(' Primary: (%6.2f,%6.2f)  Secondary: (%6.2f,%6.2f)\n'
305         %(output[1][1],output[1][0],output[2][1],output[2][0]))
306     out.write(' Primary PSF:   %2i f: %9.4f (%5.2f%%)\n'
307         %(output[3],output[4],output[4]/output[7]*100))
308     out.write(' Secondary PSF: %2i f: %9.4f (%5.2f%%)\n'
309         %(output[5],output[6],output[6]/output[7]*100))
310     out.write(' Position Angle: %6.2f Separation: %6.4f\n'
311         %(output[9],output[10]))
312     out.write(' Primary Magnitude: %6.3f Secondary Magnitude: %6.3f\n'
313         %(output[11],output[12]))
314 out.close()
315 print('end')
```

A.2 Functions.py

```
1  import numpy as np
2
3  # Takes an array and bins it by nbinx in the x direction
4  # and by nbiny in the y direction.
5  def binsection (array,nbinx,nbiny):
6      temp=array.reshape(int(array.shape[0]/nbiny),nbiny,
7                         int(array.shape[1]/nbinx),nbinx)
8      temp=temp.sum(axis=(3,1))
9      return temp
10
11 # Compares each slice of array3d (the models) to realpsf (the data)
12 # The model that produces the smallest chi^2 value is returned
13 def minchi(array3d,realpsf,flux,w):
14     chi=[]
15     cx=int((array3d[0].shape[1]-1)/2)
16     cy=int((array3d[0].shape[0]-1)/2)
17     for psf_array in array3d:
18         temp_array = psf_array[cy-2:cy+3,cx-2:cx+3]*flux
19         chi.append(np.sum( (w*(realpsf-temp_array)**2)/temp_array ))
20     return chi.index(min(chi))
21
22 # Finds a flux value for the model that minimizes chi^2
23 def fitpsf1(modpsf,realpsf,base,w):
24     fluxes=[]
25     chi=[]
26     cx=int((modpsf.shape[1]-1)/2)
27     cy=int((modpsf.shape[0]-1)/2)
28     mid_mod = modpsf[cy-2:cy+3,cx-2:cx+3]
29     for scale in np.arange(0.0001,10.0001,0.0001):
```

```
30            temp_array = mid_mod*scale*base
31            fluxes.append(scale*base)
32            chi.append(np.sum( (w*(realpsf-temp_array)**2)/temp_array ))
33        return (min(chi),fluxes[chi.index(min(chi))])
34
35    # Returns three arrays, w1 has 1s in the central nine pixels and
36    # 0s along the outside.  w2 has 1s centered around a given center
37    # coordinate for the secondary.  w has 1s where w1 and w2 have 1s
38    # and has 0s everywhere else.
39    def binarybase(pcoords,scoords):
40        w1=np.zeros((5,5))
41        w2=np.zeros((5,5))
42
43        w1[pcoords[0]-1,pcoords[1]-1]=1
44        w1[pcoords[0]-1,pcoords[1]]=1
45        w1[pcoords[0]-1,pcoords[1]+1]=1
46        w1[pcoords[0],pcoords[1]-1]=1
47        w1[pcoords[0],pcoords[1]]=1
48        w1[pcoords[0],pcoords[1]+1]=1
49        w1[pcoords[0]+1,pcoords[1]-1]=1
50        w1[pcoords[0]+1,pcoords[1]]=1
51        w1[pcoords[0]+1,pcoords[1]+1]=1
52
53        w2[scoords[0]-1,scoords[1]-1]=1
54        w2[scoords[0]-1,scoords[1]]=1
55        w2[scoords[0]-1,scoords[1]+1]=1
56        w2[scoords[0],scoords[1]-1]=1
57        w2[scoords[0],scoords[1]]=1
58        w2[scoords[0],scoords[1]+1]=1
59        w2[scoords[0]+1,scoords[1]-1]=1
```

```
60      w2[scoords[0]+1,scoords[1]]=1
61      w2[scoords[0]+1,scoords[1]+1]=1
62
63      w=np.logical_or(w1,w2).astype(int)
64      return w,w1,w2
65
66  # Find the two PSF models that together make the best binary fit
67  # Their relative flux is allowed to vary in increments of 5%.
68  # Return the index id of the best primary and secondary.
69  def twopsf(array3d,scoords,realpsf,flux,w):
70      chi=[]
71      for array1 in array3d:
72          cx=int((array1.shape[1]-1)/2)
73          cy=int((array1.shape[0]-1)/2)
74          psf1 = array1[cy-2:cy+3,cx-2:cx+3]
75          for array2 in array3d:
76              psf2=array2[cy-scoords[0]:cy-scoords[0]+5,cx-scoords[1]:cx-scoords
                    [1]+5]
77              for relflux in np.arange(0.95,0,-.05):
78                  temp_array=(psf1*relflux+psf2*(1-relflux))*flux
79                  temp=np.sum(((w*(realpsf-temp_array))**2)/temp_array)
80                  chi.append(temp)
81      chi = np.asarray(chi).reshape(100,100,19)
82      tout=np.unravel_index(chi.argmin(), chi.shape)
83      return tout[0:2]
84
85  # Search in and around the before calculated best primary and secondary,
86  # allowing to deviate by 0.1 pixel max for each.  Find a more refined
87  # relative flux, using increments of 1%.
88  # Return the best primary, secondary and relative flux.
```

```
89  def tworef(arrays,scoords,realpsf,flux,w,bestprim,bestsec):
90      if bestprim < 10:
91          bestprim+=10
92      if bestprim >= 90:
93          bestprim-=10
94      if bestprim%10 == 0:
95          bestprim+=1
96      if bestprim%10 == 9:
97          bestprim-=1
98
99      if bestsec < 10:
100         bestsec+=10
101     if bestsec >= 90:
102         bestsec-=10
103     if bestsec%10 == 0:
104         bestsec+=1
105     if bestsec%10 == 9:
106         bestsec-=1
107
108     refprim=[bestprim-11]
109     refprim.append(bestprim-10)
110     refprim.append(bestprim-9)
111     refprim.append(bestprim-1)
112     refprim.append(bestprim)
113     refprim.append(bestprim+1)
114     refprim.append(bestprim+9)
115     refprim.append(bestprim+10)
116     refprim.append(bestprim+11)
117
118     refsec=[bestsec-11]
```

```
119         refsec.append(bestsec-10)
120         refsec.append(bestsec-9)
121         refsec.append(bestsec-1)
122         refsec.append(bestsec)
123         refsec.append(bestsec+1)
124         refsec.append(bestsec+9)
125         refsec.append(bestsec+10)
126         refsec.append(bestsec+11)
127
128         chi=[]
129         for prim in refprim:
130             cx=int((arrays[prim].shape[1]-1)/2)
131             cy=int((arrays[prim].shape[0]-1)/2)
132             psf1 = arrays[prim][cy-2:cy+3,cx-2:cx+3]
133             for sec in refsec:
134                 psf2=arrays[sec][cy-scoords[0]:cy-scoords[0]+5,cx-scoords[1]:cx-
                        scoords[1]+5]
135                 for relflux in np.arange(0.99,0,-.01):
136                     temp_array=(psf1*relflux+psf2*(1-relflux))*flux
137                     temp=np.sum(((w*(realpsf-temp_array))**2)/temp_array)
138                     chi.append(temp)
139         chi = np.asarray(chi).reshape(9,9,99)
140         tout=np.unravel_index(chi.argmin(), chi.shape)
141         return (refprim[tout[0]],refsec[tout[1]],np.arange(0.99,0,-.01)[tout[2]])
142
143 # Once a best primary and secondary model are chosen and a relative flux
144 # is set, this function will search for a total flux that best fits the data.
145 def fitpsf2(primary,sec,scoords,realpsf,base,w,relflux):
146     cx=int((primary.shape[1]-1)/2)
147     cy=int((primary.shape[0]-1)/2)
```

```
148     psf1 = primary[cy-2:cy+3,cx-2:cx+3]
149     psf2 = sec[cy-scoords[0]:cy-scoords[0]+5,cx-scoords[1]:cx-scoords[1]+5]
150     fluxes=[]
151     chi=[]
152     sumpsf=psf1*relflux+psf2*(1-relflux)
153     for scale in np.arange(0.01,10.01,0.01):
154         scalepsf=sumpsf*scale*base
155         fluxes.append(scale*base)
156         chi.append(np.sum((w*(realpsf-scalepsf)**2)/scalepsf))
157     return fluxes[chi.index(min(chi))]
158
159 # This function will initialize the flux before the while loop
160 def scpsf2(primary,sec,scoords,w2,realpsf,flux,relflux):
161     cx=int((primary.shape[1]-1)/2)
162     cy=int((primary.shape[0]-1)/2)
163     psf1 = primary[cy-2:cy+3,cx-2:cx+3]
164     psf2 = sec[cy-scoords[0]:cy-scoords[0]+5,cx-scoords[1]:cx-scoords[1]+5]
165     fluxes=[]
166     chi=[]
167     rpsf2=w2*(realpsf-psf1*flux*relflux)
168     for scale in np.arange(0.0001,10.0001,.0001):
169         scalepsf2=w2*psf2*scale*flux
170         fluxes.append(scale*flux)
171         chi.append(np.sum((w2*(rpsf2-scalepsf2)**2)/scalepsf2))
172     return fluxes[chi.index(min(chi))]
173
174 # This function can take the primary or secondary, and refine the
175 # flux for that element that minimizes chi^2.
176 def rfpsf(psf1,coords1,psf2,coords2,w,realpsf,base,flux2):
177     cx=int((psf1.shape[1]-1)/2)
```

```
178     cy=int((psf1.shape[0]-1)/2)
179     psf1 = psf1[cy-coords1[0]:cy-coords1[0]+5,cx-coords1[1]:cx-coords1[1]+5]
180     psf2 = psf2[cy-coords2[0]:cy-coords2[0]+5,cx-coords2[1]:cx-coords2[1]+5]
181     fluxes=[]
182     chi=[]
183     rpsf=w*(realpsf-psf2*flux2)
184     for scale in np.arange(0.0001,10.0001,.0001):
185         scalepsf=w*psf1*scale*base
186         fluxes.append(scale*base)
187         chi.append(np.sum((w*(rpsf-scalepsf)**2)/scalepsf))
188     return (fluxes[chi.index(min(chi))])
```

A.3 Cameras.json

```
{
  "NICMOS1":
  {
    "CameraName": "NICMOS1",
    "Telescope": "HST",
    "Background":  0.0,
    "Aperture":  11.5,
    "PlateScaleX": 0.0431237,
    "PlateScaleY": 0.0429497,
    "Filters":
    {
     "F090M":{"Fv":2266.0,"apcorr":1.1103},
     "F095N":{"Fv":2046.4,"apcorr":1.1157},
     "F097N":{"Fv":2275.3,"apcorr":1.1183},
     "F108N":{"Fv":1937.0,"apcorr":1.1282},
     "F110M":{"Fv":1884.3,"apcorr":1.1331},
     "F110W":{"Fv":1855.8,"apcorr":1.1369},
     "F113N":{"Fv":1820.9,"apcorr":1.1359},
     "F140W":{"Fv":1373.3,"apcorr":1.1665},
     "F145M":{"Fv":1247.2,"apcorr":1.1786},
     "F160W":{"Fv":1086.0,"apcorr":1.1854},
     "F164N":{"Fv":1003.0,"apcorr":1.1878},
     "F165M":{"Fv":1026.8,"apcorr":1.1881},
     "F166N":{"Fv":1047.6,"apcorr":1.1882},
     "F170M":{"Fv": 981.8,"apcorr":1.1899},
     "F187N":{"Fv": 803.9,"apcorr":1.1983},
     "F190N":{"Fv": 836.4,"apcorr":1.2003}
    }
  },
```

```
30    "NICMOS2":
31    {
32       "CameraName": "NICMOS2",
33       "Telescope": "HST",
34       "Background":  0.0,
35       "Aperture":  6.5,
36       "PlateScaleX": 0.0758667,
37       "PlateScaleY": 0.0751852,
38       "Filters":
39       {
40        "F110W":{"Fv":1857.7,"apcorr":1.1436},
41        "F160W":{"Fv":1086.9,"apcorr":1.1931},
42        "F165M":{"Fv":1023.9,"apcorr":1.1962},
43        "F171M":{"Fv": 966.3,"apcorr":1.1984},
44        "F180M":{"Fv": 904.1,"apcorr":1.2041},
45        "F187N":{"Fv": 805.3,"apcorr":1.2080},
46        "F187W":{"Fv": 848.7,"apcorr":1.2084},
47        "F190N":{"Fv": 835.6,"apcorr":1.2100},
48        "F204M":{"Fv": 744.3,"apcorr":1.2222},
49        "F205W":{"Fv": 736.0,"apcorr":1.2200},
50        "F207M":{"Fv": 714.1,"apcorr":1.2247},
51        "F212N":{"Fv": 691.0,"apcorr":1.2281},
52        "F215N":{"Fv": 670.5,"apcorr":1.2291},
53        "F216N":{"Fv": 631.1,"apcorr":1.2300},
54        "F222M":{"Fv": 635.1,"apcorr":1.2326},
55        "F237M":{"Fv": 570.9,"apcorr":1.2402}
56       }
57    },
58
59
```

```
60    "NICMOS3":
61    {
62      "CameraName": "NICMOS3",
63      "Telescope": "HST",
64      "Background":  0.0,
65      "Aperture":  5.5,
66      "PlateScaleX": 0.201724,
67      "PlateScaleY": 0.200985,
68      "Filters":
69      {
70       "F108N":{"Fv":1945.7,"apcorr":1.0549},
71       "F110W":{"Fv":1863.4,"apcorr":1.0562},
72       "F113N":{"Fv":1824.9,"apcorr":1.0577},
73       "F150W":{"Fv":1218.5,"apcorr":1.0806},
74       "F160W":{"Fv":1085.7,"apcorr":1.0873},
75       "F164N":{"Fv":1003.1,"apcorr":1.0899},
76       "F166N":{"Fv":1048.9,"apcorr":1.0904},
77       "F175W":{"Fv": 967.9,"apcorr":1.0938},
78       "F187N":{"Fv": 803.8,"apcorr":1.1010},
79       "F190N":{"Fv": 835.6,"apcorr":1.1021},
80       "F196N":{"Fv": 786.6,"apcorr":1.1044},
81       "F200N":{"Fv": 768.4,"apcorr":1.1056},
82       "F212N":{"Fv": 690.9,"apcorr":1.1120},
83       "F215N":{"Fv": 670.6,"apcorr":1.1136},
84       "F222M":{"Fv": 635.1,"apcorr":1.1187},
85       "F240M":{"Fv": 559.9,"apcorr":1.1328}
86      }
87    }
88  }
```