PROCESSING OF SIMULATED AND EXPERIMENTAL IMAGES OF CLOSELY SPACED BINARY STARS USING SPECKLE INTERFEROMETRY

A Thesis

presented to

the Faculty of California Polytechnic State University,

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Electrical Engineering

by

Niels Christian Smidth

June 2016
COMMITTEE MEMBERSHIP

TITLE: Processing of Simulated and Experimental Images of Closely Spaced Binary Stars Using Speckle Interferometry

AUTHOR: Niels Christian Smidth

DATE SUBMITTED: June 2016

COMMITTEE CHAIR: John Ridgely, Ph.D.
Professor of Mechanical Engineering

COMMITTEE MEMBER: Russell Genet, Ph.D.
Research Scholar in Residence

COMMITTEE MEMBER: Jane Zhang, Ph.D.
Professor of Electrical Engineering
ABSTRACT

Processing of Simulated and Experimental Images of Closely Spaced Binary Stars Using Speckle Interferometry

Niels Christian Smidt

Theory and methods of processing speckle interferometry data from close visual binary stars are presented and implemented. The effects of the optical systems used for observing close visual binary stars are explained and simulated from both the geometrical and physical optical viewpoints. The atmospheric phase distortion and shot noise responsible for the observed speckle patterns are simulated. The deconvolution technique originally presented by Labeyrie is implemented to extract astrometric data from close visual binary stars. This method is applied to both simulated and experimental data from Kitt Peak National Observatory as validation. Parts of the deconvolution process are optimized to allow for near real time calculations in an automated observatory.
ACKNOWLEDGMENTS

I would like to thank my advisor, John Ridgely, for all the time he has spent on this project while on sabbatical. In addition, I have greatly appreciated Russell Genet’s countless hours spent getting me up to speed as well as for his unrelenting passion to the advancement of astronomy and getting students involved with research. A big thanks to David Rowe as well, who was instrumental in helping my understanding of some of the most difficult to understand concepts of this project. Lastly, I have to express my gratitude to my family, who have have been incredibly helpful in both technical and moral support.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>LIST OF FIGURES</th>
<th>........................................</th>
<th>viii</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHAPTER</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 INTRODUCTION</td>
<td>........................................</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Objective</td>
<td>........................................</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Motivation</td>
<td>........................................</td>
<td>1</td>
</tr>
<tr>
<td>2 BACKGROUND</td>
<td>........................................</td>
<td>3</td>
</tr>
<tr>
<td>2.1 Binary Stars</td>
<td>.......................................</td>
<td>3</td>
</tr>
<tr>
<td>2.2 Telescope Theory</td>
<td>.....................................</td>
<td>4</td>
</tr>
<tr>
<td>2.3 Optical Transfer Function and Fourier Optics</td>
<td>..................................</td>
<td>10</td>
</tr>
<tr>
<td>2.4 Telescope Seeing Limit</td>
<td>..................................</td>
<td>14</td>
</tr>
<tr>
<td>2.5 Atmospheric Distortion</td>
<td>..................................</td>
<td>18</td>
</tr>
<tr>
<td>3 LABEYRIE DECONVOLUTION</td>
<td>...................................</td>
<td>26</td>
</tr>
<tr>
<td>3.1 Labeyrie's Findings</td>
<td>..................................</td>
<td>26</td>
</tr>
<tr>
<td>3.2 Theory</td>
<td>........................................</td>
<td>27</td>
</tr>
<tr>
<td>3.3 Implementation</td>
<td>......................................</td>
<td>44</td>
</tr>
<tr>
<td>4 IMPLEMENTATION IN AUTOMATED OBSERVATORY</td>
<td>.........................</td>
<td>46</td>
</tr>
<tr>
<td>4.1 Optimization</td>
<td>......................................</td>
<td>46</td>
</tr>
<tr>
<td>4.2 Implementation</td>
<td>......................................</td>
<td>48</td>
</tr>
<tr>
<td>5 FUTURE WORK</td>
<td>........................................</td>
<td>53</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>The orbits of a binary star pair [1]</td>
<td>3</td>
</tr>
<tr>
<td>2.2</td>
<td>Ray trace diagram of a prime focus telescope</td>
<td>5</td>
</tr>
<tr>
<td>2.3</td>
<td>Ray trace diagram sign convention</td>
<td>6</td>
</tr>
<tr>
<td>2.4</td>
<td>Model of a celestial object as point source</td>
<td>7</td>
</tr>
<tr>
<td>2.5</td>
<td>Kitt Peak 2.1m mirror simulation</td>
<td>9</td>
</tr>
<tr>
<td>2.6</td>
<td>Behavior of light through an aperture [2]</td>
<td>11</td>
</tr>
<tr>
<td>2.7</td>
<td>Relationship between spatial and spatial frequency domains</td>
<td>12</td>
</tr>
<tr>
<td>2.8</td>
<td>Using the Fraunhofer diffraction to calculate PSF of pupil</td>
<td>14</td>
</tr>
<tr>
<td>2.9</td>
<td>KPNO 2.1m telescope PSF (logarithmic intensity)</td>
<td>16</td>
</tr>
<tr>
<td>2.10</td>
<td>Diffraction limited binary star simulation</td>
<td>16</td>
</tr>
<tr>
<td>2.11</td>
<td>Center row of aperture diffraction image</td>
<td>17</td>
</tr>
<tr>
<td>2.12</td>
<td>Minimum spacing as defined by Rayleigh Criterion [3]</td>
<td>17</td>
</tr>
<tr>
<td>2.13</td>
<td>Creation of atmospheric turbulence [4]</td>
<td>18</td>
</tr>
<tr>
<td>2.14</td>
<td>Atmospheric distortion's effect on telescope images [5]</td>
<td>20</td>
</tr>
<tr>
<td>2.15</td>
<td>Phase and magnitude of complex pupil image</td>
<td>21</td>
</tr>
<tr>
<td>2.16</td>
<td>Simulated PSF through atmospheric distortion</td>
<td>21</td>
</tr>
<tr>
<td>2.17</td>
<td>Adding shot noise to simulated image</td>
<td>23</td>
</tr>
<tr>
<td>2.18</td>
<td>Adding Gaussian read noise to simulated image</td>
<td>25</td>
</tr>
<tr>
<td>3.1</td>
<td>Short vs long exposure times through atmospheric distortion</td>
<td>26</td>
</tr>
<tr>
<td>3.2</td>
<td>Average PSD and autocorrelation of binary star images</td>
<td>28</td>
</tr>
</tbody>
</table>
1. INTRODUCTION
1.1 Objective

Observing binary stars using speckle interferometry requires software tools to extract the astrometric measurements. This project is concerned with presenting research on the theory behind speckle interferometry as well as creating well documented speckle interferometry software tools in a modern, cross platform language that allow for continual development by further students.

1.2 Motivation

The theory behind the observation of close visual binary stars with speckle interferometry is a multi-disciplinary task, spanning the fields of physics, Fourier optics, image processing, and computation. Creating simulations of the phenomena behind speckle interferometry allows students more knowledgeable in a single field of the process to understand the entire process and work with experts of the other fields.

Observing closely spaced binary stars requires a large telescope, which provides a high enough angular resolution to see the two distinct targets. Getting observing time on a 3-meter or greater telescope required to make these observations is quite difficult for academic groups today. To allow for academic groups to easily perform double star observations using speckle interferometry, the construction of a suitable automated telescope has been proposed by Russell Genet et al. [6]. This proposed 4-meter class telescope has been estimated to cost nearly a
hundredth of what a conventional telescope of this class might cost. Part of this project involves creating a software tool chain allowing students from different academic institutions to make observations and get processed data remotely. Like the design of the telescope, this software should be modular and well documented so that it may be modified by students and instructors as the scope of the project changes over time.
2. BACKGROUND

2.1 Binary Stars

It is estimated that up to two thirds of solar stars nearby have stellar companions [7]. Many of these systems may be described as binary stars, a pair of gravitationally bound stars. Each star has its own elliptical orbit around the common focus known as the barycenter, the pair’s common center of mass, as seen in Figure 2.1. Binary stars are of great interest to astronomers as their visible interaction with each other gives the opportunity to calculate the mass of the total system and the individual stars. The mass of a star gives important insight into the lifecycle of the star, making it valuable in research on the evolution of the universe and modeling stellar behavior.

![Orbits of Stars in a Binary System](image)

Figure 2.1: The orbits of a binary star pair [1]

Many of these binary star pairs are either too far away from Earth or too closely spaced from each other to be able to visually resolved. These may be studied using techniques such as spectroscopy to view the Doppler shifts spectrum from radial velocity changes. This thesis is concerned with visual binary stars, which are spaced far apart enough to be visually resolved with a telescope. Using telescopes,
the separation and position angle of the orbit may be observed over time, allowing the use of Kepler’s laws of planetary motion to calculate masses.

2.2 Telescope Theory

When viewing a very distant object with the unaided eye, its projection onto the retina is very small compared to the area of the retina. This small projection covers very few rods/cones on the retina, which may be compared to the pixels on a digital camera’s CCD. Because only a few pixels receive the projection, we are unable to make out significant detail on the distant object. For example, a binary star appears as a bright point in the sky to the naked eye, rather than two separate visible spheres. Due to the unaided eye’s inability to study such distant objects, we use a telescope to collect and concentrate the light from distant objects onto an imaging sensor, which may be the eye or a CCD sensor. The ability of a telescope with an image sensor to form a usable image of a binary star may be explained with concepts from geometrical optics.

The telescopes commonly used in viewing visual binary stars are variations on a basic reflecting telescope design. A reflecting telescope uses one or several shaped mirrors to focus the incoming light from the target into an image. The simplest single mirror reflecting telescope is the prime focus telescope, depicted in Figure 2.2. A prime focus telescope uses a single parabolic mirror to focus incoming parallel light rays into a single focal point [3]. Important dimensions of this mirror are its focal length and diameter, also depicted in Figure 2.2.
For this discussion of basic functioning of a telescope, we can consider these primary mirrors to be in the size range of 0.5 meters, the size of a serious amateur telescope, all the way up to 10m, the size of one of the two Keck telescopes on Mauna Kea. These telescope mirrors are many magnitudes larger than the wavelength of visible light, in the band of 390nm-700nm. This means that the wave properties of light such as diffraction and interference occur on a scale considerably smaller than the mirrors, so we may approximate light as rays traveling in a straight line interacting with geometrical optics [8]. For example: in Figure 2.2, the parabolic mirror’s ability to focus incoming parallel light rays is shown using the geometrical optics approximation.

In order to simplify the analysis of these geometrical optics, some approximations may be made. If the incident and reflected light rays (see Figure 2.3) are
at a small angle with respect to the optical axis, the paraxial approximation can be made [9]. The paraxial approximation allows the following simplifications to be made:

\[ \sin(\alpha) = \alpha \quad \cos(\alpha) = 1 \quad \tan(\alpha) = \alpha \]

These simplifications make calculating reflected angles much easier, as well as allowing the use of ray transfer matrices for calculating incident and reflected angles of light rays.

The ray transfer matrix method is used to calculate the height and angle of reflected/refracted rays given the shape of the interface as well as the height and angle of the incident rays. The general form of the matrix is:

\[
\begin{bmatrix}
\alpha_{\text{reflected}} \\
x_{\text{reflected}}
\end{bmatrix}
= 
\begin{bmatrix}
A & B \\
C & D
\end{bmatrix}
\begin{bmatrix}
\alpha_{\text{incident}} \\
x_{\text{incident}}
\end{bmatrix}
\]

The values in the ABCD matrix are related to the interface the light rays interact with. For a parabolic concave mirror, the ray transfer matrix is defined as [10]:

Figure 2.3: Ray trace diagram sign convention
This ray transfer matrix is used in a simulation (See Appendix 6) of the Kitt Peak National Observatory 2.1m telescope to show the parabolic mirror’s ability to focus parallel light rays incident at different angles. The results of this simulation are shown in 2.5. When mirrors with a focal length several times greater than mirror diameter are used, reflected angles are small and the paraxial approximation remains valid. In reality, comatic aberration, which causes distortion of the image, increases with increasing incident ray angle [4]. This may be ignored for this discussion as the telescope configurations used in binary star observations typically have a small field of view, and thus small light ray angles, to give high detail when viewing objects with small features.

Distant objects like stars may be modeled as point sources of light, giving off light rays evenly in every direction. Because these stars are at an effectively infinite
distance, these light rays become effectively parallel by the time they reach the earth, as shown in Figure 2.4 [4]. This behavior may be applied to all distant celestial objects in our field of view, each emitting light that becomes a parallel set of rays by the time they reach the earth.

The difference between the parallel rays from these different sources is their incident angles. As shown in Figure 2.5, each parallel bundle of light rays from a specific angle is focused to a different unique point. The image formed on an image sensor is the superposition of all the different angled parallel light ray bundles, each focused down to a point on the image sensor.

This knowledge allows for calculation of the image scale of a telescope, which gives the relation between subtended angles in the sky and distances on the image sensor. With this knowledge, the field of view and pixel scales may be calculated as well [4]. These quantities allow for converting between units in the image plane and units in the sky.

\[
\text{image scale}[\text{arcsec}/m] = \frac{206265[\text{arcsec}]}{\text{focal length}[m]}
\]

\[
\text{field of view}[\text{arcsec}] = (\text{image scale}[\text{arcsec}/m]) \times (\text{image sensor width}[m])
\]

\[
\text{pixel scale}[\text{arcsec/pixel}] = (\text{image scale}[\text{arcsec}/m]) \times (\text{pixel width}[m])
\]

The ability of a telescope to form the image of a celestial scene was explained using geometrical optics. With this understanding, a binary star would reduce down to two perfect points of light on the image sensor, giving the viewer the ability to precisely measure the angle and distance between the stars. The Andor Luca...
R EMCCD camera, used by Russell Genet et al in their binary star observations at Kitt Peak Observatory [11], has a pixel size of only $8\mu m \times 8\mu m$ [12]. Images captured with this much detail will reveal the more complex interference effects of
light, making the geometrical optics model not accurate enough. In order to fully model the behavior of the telescope and image sensor combination, techniques involving the wave nature of light must be used.

2.3 Optical Transfer Function and Fourier Optics

When considering light in geometrical optics, rays are used to show the transmission of light through interfaces such as lenses and apertures. Each ray follows straight lines, changing direction when transmitting through different materials (refraction) or encountering a reflective surface (reflection). This knowledge allowed for the use of ray trace matrices in showing the behavior of an optical system. When rays pass through an aperture, those outside the diameter of the aperture are stopped and absorbed, while those within the diameter pass through unaffected. When considering the electromagnetic wave behavior of light, an interface such as a lens causes change in the wave’s phase at each spatial coordinate, resulting in the changing of the wavefront’s direction. An aperture now causes a diffraction pattern to be visible in the light intensity field produced on a screen after the aperture. The difference between wave and ray consideration of light with an aperture is shown in Figure 2.6. So in wave optics, an interface is modeled by considering the changes that incident light undergoes in magnitude and phase at each spatial point.

When considering the wave nature of light, optical systems may be analyzed with similar techniques as are employed when modeling responses of circuits or mass-spring-damper systems to an input [3]. The signals involved in optical sys-
tems are two-dimensional spatial signals, rather than the one-dimensional temporal signals commonly used when modeling systems. An optical system may be modeled by its Point Spread Function (PSF), the intensity pattern produced in response to the input of a single point of light. The optical system may also be modeled in the spatial frequency domain by its Optical Transfer Function (OTF), which forms a Fourier pair with the PSF. The response of an optical system to an input may be calculated as shown in 2.7.

It is important to note that these responses are only valid for a source that is spatially and temporally coherent. Coherence refers to the strength of correlation between signals at different times or locations and is related to the ability to produce interference effects [13] [14]. A temporally coherent source has little random fluctuation in wavefront spacing over time. This means that at a single observing
point but different times, observed light over a time interval will exhibit high levels of correlation. A spatially coherent source has little variation in wavefront spacing at different points. This means that at different observing points but at the same time, observed light over a time interval will exhibit high levels of correlation. An example of a near ideal source (both spatially and temporally coherent) is a laser, which is very near an ideal point source (making it spatially coherent) and produces nearly monochromatic light (making it temporally coherent). The sun is neither spatially nor temporally coherent, as it is considerably larger than a point source, even to the naked eye, and is a very wideband source. The binary stars under observation are considerably farther away than the sun, making them effectively point sources (and therefore spatially coherent). Using a narrowband filter in the observing telescope makes the binary star appear as temporally coherent as well. Since filtered binary stars may be considered as temporally and spatially coherent sources, this OTF analysis may be applied to their observation. For spatially incoherent sources, this OTF analysis may no longer be applied. The lack of coherence prevents interfer-
ence, the output signal is defined as the superposition of intensity images (rather
than the superposition of phasors for coherent imaging) [14].

The prime focus telescope modeled previously using geometrical optics may
also be fully modeled using wave optics, but this is a large task and most of the
important behavior of this telescope due to the wave nature of light may be shown
using some approximations. The dominant diffraction effect in a telescope is due to
the finite size of optical components. At the far field distance, where the optical path
length differences from points at dimensional extremes of an optical component is
significantly less than the wavelength, the Fraunhofer diffraction pattern is gener-
at [15]. The distance at which this diffraction pattern occurs can be estimated in
several ways, one common method is the "antenna designer's formula" [16]:

\[ z > \frac{2D^2}{\lambda} \]

where \( z \) is the separation between the plane of the optical component and the
beginning of the far field, \( D \) is the largest diameter of the optical component, and
\( \lambda \) is the wavelength of the light.

The primary parabolic lens or mirror of a telescope may be modeled as a finite
sized aperture (with \( D \) = diameter of primary lens) over a larger lens, so incoming
light outside of \( D \) is not transmitted. As an example, for 0.6\( \mu \)m red light with a 2.1m
aperture diameter, the far field occurs at \( 14.7 \times 10^6 \) meters away from the aperture.
Distances of this amplitude are considerably larger than the lengths of most tele-
scopes. However, the parabolic lens focuses this light, which would normally show
the Fraunhofer diffraction at a very large distance, at its focal length. Thus in a
simple telescope system, the Fraunhofer diffraction pattern of the primary lens is shown on the image sensor.

Under these conditions, the complex diffraction pattern at the focal plane is calculated as the Fourier transform of the complex valued pupil image. In reality, there are other phase terms and higher order terms in the diffraction pattern, but for the small field of view found in the telescopes used in speckle interferometry, these may be ignored. This may be used to calculate the PSF of different components of the optical system as shown in Figure 2.8. These techniques will be used to analyze the response of a simple Prime Focus telescope as well as the effects of atmospheric distortion.

![Diagram showing the relationship between Complex Aperture, OTF, Field Pattern, and PSF](image)

**Figure 2.8:** Using the Fraunhofer diffraction to calculate PSF of pupil

### 2.4 Telescope Seeing Limit

Using geometrical optics analysis, a prime focus telescope will produce a perfect image of a star on the image sensor, which will appear as a tiny point/circle. Taking
into account the wave model of light, the more accurate image produced may be calculated.

The OTF technique illustrated in Figure 2.7 is utilized to simulate the Kitt Peak National Observatory’s (KPNO) 2.1m telescope, as seen by the Andor Luca-R camera (Appendix B). This simulation uses a Fast Fourier Transform rather than a continuous Fourier Transform to calculate the PSF of the pupil function. The PSF of this telescope’s main mirror (as seen in an Andor Luca-R camera) is shown in 2.9. This PSF shape, resulting from the diffraction pattern of a circular aperture (with magnitude = 1 inside the aperture, magnitude = 0 outside the aperture), is known as an Airy Disk. This Airy disk is shown in a logarithmic intensity scale to accentuate the fringes that surround the central peak. A simulated binary star and the simulated output image is shown in Figure 2.10. The output image is explained as two Airy Disks, each superposed at the location of the point source input.

The effects of diffraction on the output image of a binary star can be more easily viewed in one dimension in Figure 2.11. In one dimension, the Airy disk bears a strong resemblance to the sinc function. The first null of this function occurs in the image sensor plane at a distance from center of

\[ \text{radius} = 1.22 \cdot \lambda \cdot \frac{\text{focal length}}{\text{focal length} \cdot \text{focal length}} \]

The Rayleigh Criterion states that this radius is the minimum distance between two distinguishable point objects in an optical system. This distance is shown in 2.12. As the distance between two point objects gets smaller than this distance, the central peak of the two Airy disks begin to smudge together, preventing iden-
Figure 2.9: KPNO 2.1m telescope PSF (logarithmic intensity)

Figure 2.10: Diffraction limited binary star simulation

tification of each individual peak's location. In order to resolve objects with fine
Figure 2.11: Center row of aperture diffraction image

details, a larger diameter primary mirror is required to decrease the size of the Airy disk.

Figure 2.12: Minimum spacing as defined by Rayleigh Criterion [3]

These figures and simulations show that an ideal telescope has a limited angular resolution and that detail finer than this resolution will be lost. This angular resolution may be improved by increasing the diameter of the telescope. This implies that the diameter of the telescope may be increased indefinitely to increase the angular resolution of a telescope, allowing for studying of infinitely small objects. Unfortunately for earthbound telescopes, the distortion caused by variable density air pockets in the atmosphere often prevents reaching the theoretical diffraction limit on a large telescope.
2.5 Atmospheric Distortion

All previous explanations and simulations of telescope behavior assume that the light incident on the telescope from its target is a perfect plane wave, with a flat wavefront. For space telescopes outside of the Earth’s atmosphere this assumption is quite close to reality, but for telescopes on Earth’s surface, beneath the atmosphere, this assumption no longer holds. Space telescopes, such as the Hubble Space Telescope, are able to create images at quite close to the diffraction limited resolution shown in the previous section. Transmitting through the atmosphere, the flat wavefront of an incident plane wave is turned into a nonuniform, bumpy wavefront that will no longer be imaged as a single, stationary Airy disk by a telescope on Earth.

Due to the sun’s heating of the surface of the earth, convective cells are created in the lower atmosphere. At the boundaries of these cells, shear forces between columns of air with different velocities create turbulence, depicted in Figure 2.13. This turbulence creates cells of variable density and temperature air. Both density and temperature affect the index of refraction of a medium, so this atmospheric turbulence creates cells of variable index of refraction which light from space must propagate through to be seen by a telescope on Earth.

Figure 2.13: Creation of atmospheric turbulence [4]
Having a spatially variable index of refraction means that the phase shift experienced by an incident flat wavefront is now spatially variable as well, making the transmitted wavefront no longer flat. The phase distortion of the wavefront may be numerically classified with Fried's parameter, the expected diameter over which the root-mean-square optical phase distortion is equal to one radian. From a mountaintop observatory in good seeing conditions, the Fried parameter is approximated as between 10-20cm. Within this diameter the wavefront may be approximated as flat [4]. If the main lens/mirror of a telescope is smaller than this diameter, an approximately flat wavefront will be received by the telescope, but this flat wavefront’s incident angle will be constantly changing. This results in a small point image of the target (due to a relatively flat waveform) that is constantly moving with time (due to a time varying incident angle). This is the case when viewing stars with the naked eye, the star appears as a point source but “twinkles”, showing its time varying incident angle. If the main lens/mirror of a telescope is considerably larger than this diameter, the received wavefront can’t be approximated as flat, but its incident angle is relatively constant with time. This results in a more distorted image that changes with time (due to the time varying incident waveform phase shift) but remains relatively stationary in location in the image (due to constant incident angle). This is the effect that creates the speckle images that this project is concerned with. These effects are shown in Figure 2.14.

To simulate these speckle images, the atmospheric distortion must be modeled in a form compatible with the previous Fourier optics simulation. Since the incident wavefront experiences spatially varying phase shift when transmitting through the
atmosphere, the atmospheric distortion may be modeled as a phase screen. The PSF of this atmospheric screen (along with the telescope aperture) may be calculated and convolved with the input image to simulate a speckle image. The atmosphere’s phase shift is modeled as a Gaussian random process with a power spectrum defined by Kolmogorov [17] [18] [19] [20]:

\[
\Phi(k) = 0.023 r_0^{-5/3} |k|^{-11/3}
\]

The same KPNO 2.1m telescope was simulated (Appendix C) with atmospheric distortion. The complex pupil image, combining the aperture and atmospheric phase shift, is shown in Figure 2.15. The magnitude portion is the same aperture image as the previous simulation, and the phase portion is the "cloudy" atmospheric distortion. The PSF of this telescope/atmospheric distortion optical system is shown in Figure 2.16:
Figure 2.15: Phase and magnitude of complex pupil image

Figure 2.16: Simulated PSF through atmospheric distortion

This simulation PSF image represents a very short time exposure of a single star. In this speckle image, the size of the small features (the speckles) are affected
by the size of the lens, while the overall envelope of the speckle pattern is affected by the atmospheric conditions. As atmospheric seeing improves, Fried's parameter increases, and this envelope decreases in size.

When observing with fast exposures on dim targets, the number of photons detected during an exposure may become low enough that photon shot noise becomes an important noise source. The arrival of a photon from a star source is a temporally random, discrete event, with each event independent from the others [21] [22]. For a dim target/short exposure, where the number of photons arriving from the source is small, this random element of the captured signal becomes very visible, causing noticeable differences between subsequent images of the same source. For a bright target/long exposure, the number of photons arriving is very large, so the random element becomes less noticeable and subsequent images are nearly the exact same.

The probability of a number of photons being captured over a time interval is modeled with a Poisson distribution:

\[
P(N) = \frac{e^{-\lambda} \lambda^N}{N!}
\]

Where: \(N = \text{Photons Detected}; \lambda = \text{Expected Photons Detected}

In a simulation not taking shot noise (or other noise sources) into account, subsequent images of a static object should be exactly the same, meaning the number of photons received at each pixel of the photo sensor are the exact same between images. So to add shot noise to a simulation like this, each pixel's new value may be calculated as a random value from a Poisson distribution, whose expected pho-
tons detected, $\lambda$, value corresponds to the old, static pixel value. If all the old pixel values for the entire sensor are added together, this gives the total number of photons expected to arrive at the sensor. Scaling the entire image by a constant scales the number of photons appearing in the new shot noise image.

An important part of considering noise in simulations and measurements is knowing when to consider the effects of the noise significant or not. Because this noise is modeled as a Poisson distribution, the relationship between the standard deviation and mean are known as $\sigma^2 = \mu$ [4]. This means the signal to noise ratio may be calculated as:

$$SNR = \frac{\mu}{\sigma} = \frac{N}{\sqrt{N}} = \sqrt{N}$$

This shows that as the number of photons received increases, the SNR increases, and the effects of noise are less visible. So for suitably bright objects, this effect may be disregarded. This may be seen in the simulation image shown in Figure 2.17, comparing images from both the noise free image and noisy images with different numbers of photons received.

Figure 2.17: Adding shot noise to simulated image
For 1000 photons detected, the image barely resembles its uncorrupted version, indicating a low SNR. But for 100E3 photons incident, the image begins to look very similar to the uncorrupted version, indicating a higher SNR. A degraded SNR may limit the precision of measurements made by the astronomer, so there is a fundamental limit to how dim the target may be or how fast the shutter speed may be.

In addition to photon shot noise, there are several additive noise sources that may contribute to the final image such as CCD read noise (both on and off the sensor chip) and electronic interference. These noise sources are dependent on factors such as the design of the camera/image sensor, the temperature of the sensor, and the pixel read speed. These noise sources are modeled as additive gaussian white noise functions [23] [24] [25]. That is, each pixel has an independent gaussian random value added to it. A comparison of a simulated reference star image with and without Gaussian read noise is shown in Figure 2.18, where number of electrons incident = 50 * 10^3 and additive read noise variance = 3 photons.

This theory and simulation shows that when observing with a large telescope from beneath the atmosphere, such as the KPNO 2.1m telescope, the quality of the images captured is limited by the atmospheric seeing and noise sources, rather than just the size of the primary optic. This means that for closely spaced visual binary stars, the raw images captured rarely show two distinct objects easily identified as two stars. In order to view these binary stars, speckle interferometry techniques may be employed.
Figure 2.18: Adding Gaussian read noise to simulated image
3. LABEYRIE DECONVOLUTION

3.1 Labeyrie’s Findings

Labeyrie took advantage of the interference properties of light in order to recover diffraction limited information from the image. Because his important findings were in 1970, many of his "computations" were performed optically rather than computationally.

When a close binary star is viewed with an eyepiece or a long exposure photograph, the high speed changes in the speckle pattern are integrated and blurred together, producing the large atmospheric seeing limited pattern. When the same star is viewed with a very short exposure photograph (10-15ms), some high spatial frequency data is visible in the form of the bright speckles. These differences between long and short exposure image can be seen in Figure 3.1.

![Figure 3.1: Short vs long exposure times through atmospheric distortion](image-url)
In all the previous simulations, the assumption has been made that the targets are all within the isoplanatic patch. The isoplanatic patch is defined as the area over the sky where the PSF is constant [26]. This isoplanatic patch is often estimated to be around 5 arcsec in good seeing conditions for visible light [4]. If the primary and secondary of a binary star are located within this isoplastic patch, the short exposure observed image will be a superposition of two shifted PSFs, one for each star. This can be seen in the first row of Figure 3.2. The atmospheric distortion simulations for the initial explanation of this theory leave out the noise sources to keep equations simple.

So with short exposure images of binary stars within the isoplanatic patch, higher frequency diffraction limited information is preserved and may be recovered. Labeyrie observes the Fraunhofer diffraction pattern generated by these binary star images to have interference fringes due to their inherent periodicity [27][28]. While this information is not enough to totally reconstruct the original image, Labeyrie notes that it is sufficient to extract data on the separation distance and angle of the binary star.

3.2 Theory

The explanation of the formation of these fringes can be done using the OTF theory presented in Section 2.3 [27]. Where Labeyrie used lasers and screens to create Fraunhofer diffractions, Fourier Transforms/FFTs may be used to create very similar images.
The object of study in this case is a binary star, and its spatial intensity is represented by \( o(x, y) \). The image created by the optical system viewing this object is represented as \( i(x, y) \). \( i(x, y) \) may be calculated as the convolution of the optical system’s PSF, \( p(x, y) \), with \( o(x, y) \).

\[
o(x, y) * p(x, y) = i(x, y)
\]
Viewing the Fraunhofer diffraction patterns of these images generated optically is equivalent to viewing the squared magnitude (intensity) of the Fourier Transform of these images, also known as the Power Spectral Density (PSD). The PSD of the image may be calculated as shown below, where $I(u,v), P(u,v)$, and $O(u,v)$ are the Fourier Transforms of the image, PSF, and object respectively:

$$|O(u,v)|^2 \cdot |P(u,v)|^2 = |I(u,v)|^2$$

With data from previous simulations, this relationship between the original image and their PSDs can be shown graphically in the first two rows in Figure 3.2.

If the object being observed is a binary star, the object is represented ideally as two point sources, which creates a Fraunhofer diffraction pattern of fringes. The angle and separation of these fringes hold the important astrometric information about the binary star target. These fringes are still visible in the PSD of $I(u,v)$, so the information is still present in the speckle image.

Since each image contains an element of random noise from the atmosphere, the signal to noise ratio of the PSD may be improved by averaging many image PSDs together. The atmospheric distortion is a random process that varies between each image, but some of its properties (size and number of Fried Parameter seeing cells) are relatively constant with time. These PSDs are averaged below.

Note that $< O(u,v) > = O(u,v)$ because the object is a constant.

$$|O(u,v)|^2 \cdot < |P(u,v)|^2 > = < |I(u,v)|^2 >$$

If this is all the information the astronomer has, useful information may be calculated from this PSD. If the inverse Fourier Transform is taken, the separa-
tion angle and distance of the object may sometimes be determined, depending on the quality of the seeing. The inverse Fourier Transform of a PSD function is the autocorrelation of the original spatial signal. This is shown below, where

\[ R_{ff}(x,y) = \text{autocorrelation of } f(x,y) \text{ function} \]

\[ F^{-1}\{|O(u,v)|^2 \cdot < |P(u,v)|^2 > = < |I(u,v)|^2 > \} \]

\[ R_{oo}(x,y) * < R_{pp}(x,y) >= < R_{ii}(x,y) > \]

The result of these calculations is the average autocorrelation of each binary star image. This average autocorrelation image is the convolution of the autocorrelation of the object (the desired information, constant with time) with the average autocorrelation of the aperture/atmospheric PSF (undesirable information, different in each image). This relationship is shown in the bottom row of Figure 3.2

The average binary star image autocorrelation is the most useful image from these calculations. The noticeable features are the large primary lobe in the center of the image with two smaller secondary lobes symmetrically positioned around this primary lobe. The angle and distance of the separation between the primary and secondary lobes indicates the angle and distance of the separation between the primary and secondary stars of the original binary star.

For binary stars with large enough separation angle, this average image autocorrelation can provide the astrometric information desired, as the primary and secondary lobes may be resolved. However, for closely spaced binary stars, the primary and secondary lobes begin to overlap and combine. As spacing decreases further, eventually the lobes will be impossible to resolve and the desired astromet-
ric information can no longer be gathered. To get around this problem, a reference star may be used to deconvolve the atmospheric PSF from the binary speckle images.

A reference star is a single star located within the same isoplanatic patch as the binary star is located. Observing this reference star will give the PSF of the same atmospheric distortion that the binary star is viewed through. Since the properties of this atmospheric distortion are relatively constant over time, the averaged PSD of the reference star’s PSF should be very similar to the averaged PSD of the atmospheric PSF present in binary star images. Rearranging the average image PSD equation gives:

\[ |O(u, v)|^2 = \frac{\langle |I(u, v)|^2 \rangle}{\langle |P(u, v)|^2 \rangle} \]

Since the \( \langle |P(u, v)|^2 \rangle \) term can be found by observing a reference star, this equation shows that the atmospheric distortion may be theoretically removed from the average binary star image PSD, giving the PSD of the object. This process of using the reciprocal of the degrading function is known as inverse filtering [23]. As may be seen in Figure 3.3, the deconvolved PSD calculated with this inverse filter appears as a nearly perfect set of fringes, with no shaping from the telescope aperture or atmosphere like in the average binary star image PSD. The inverse Fourier Transform of this deconvolved PSD gives the autocorrelation of the binary star object, without any of the telescope aperture or atmospheric effects shown in the binary star image autocorrelation. The result is that the lobes of the deconvolved autocorrelation are sharper and easily resolved compared to the binary star image.
autocorrelation. In fact, this deconvolved autocorrelation looks nearly identical to the input image, suggesting that the information has been entirely recovered.

Unfortunately, this simulation has been done without any noise present, which is not a realistic condition. As explained previously, both shot and gaussian noise can be expected in the images of both the binary and reference stars. This noise is not band limited (unlike the optical information capture by the telescope), but
is present at all spatial frequencies. The reference star PSD contains most of its energy in two areas, shown in Figure 3.4, where the noiseless reference star PSD is displayed logarithmically. The optical system’s components act as spatial filters, both atmospheric distortion and aperture being spatial lowpass filters. The energy transmitted by atmospheric distortion is concentrated at low frequencies, resulting in the primary, central peak in the PSD. The energy transmitted by the optical system’s finite sized aperture has a wider bandwidth and is located in the PSD’s wider lobe.

Due to this optical system’s low-pass nature, the optical energy at very high frequencies, beyond the cutoff frequency of the aperture, is much lower than the energy at lower frequencies. To make the simulation more realistic, small amounts of wideband noise are added. At low frequencies, where the majority of signal energy is located, the signal-to-noise ratio remains acceptable and isn’t significantly different. However, at very high frequencies, because of the lack of signal energy, the signal-to-noise ratio becomes very poor. When performing deconvolution through inverse filtering this can lead to this high frequency noise energy dominating low frequency signal energy in the deconvolved PSD, as shown below in Figure 3.5. This means that the signal in the deconvolved autocorrelation (seen as lobes) is dominated by noise (in the form of tall and sharp peaks), so image contrast is very low and little information can be retrieved.

This effect may be shown mathematically. The noise added to the captured image is represented as $N(u, v)$ in the frequency domain. In the presence of noise,
Figure 3.4: Sources of signal energy in noiseless PSD

the frequency domain image captured by the telescope may be described as:

\[ O(u, v) \cdot P(u, v) + N(u, v) = I(u, v) \]
If inverse filtering is used in an attempt to deconvolve the reference star from the observed binary star image, the following results:

\[
\hat{I}(u,v) = \frac{O(u,v) \cdot P(u,v) + N(u,v)}{P(u,v)} = \frac{O(u,v) \cdot P(u,v)}{P(u,v)} + \frac{N(u,v)}{P(u,v)} = O(u,v) + \frac{N(u,v)}{P(u,v)}
\]

Like before, the desired uncorrupted object \(O(u,v)\) is recovered, but this time, the wideband noise value is also present. Because the spectrum of the undesirable system PSF \(P(u,v)\) is lowpass in nature, its inverse becomes highpass shaped. This has the effect of attenuating noise at low frequencies (where the majority of the signal energy is located) and amplifying it at high frequencies (where little to no signal energy is present). This leads to the dominance of high frequency noise in the deconvolution in the presence of noise shown in Figure 3.5. This shows that when dealing with real data with noise, inverse filtering is rarely effective.
noise corrupted signal and recovery model does not perfectly apply to Labeyrie’s deconvolution, as Labeyrie’s deconvolution uses average PSDs, while this model uses the frequency domain of a single image, but the general behavior still applies.

While the inverse filter has been shown to be ineffective in the presence of noise, the Wiener filtering deconvolution technique considers noise in its derivation and still gives an accurate deconvolution estimate [23] [29]. The equation for this filter is shown below:

\[
\hat{F}(u, v) = \left[ \frac{1}{H(u, v)} \frac{|H(u, v)|^2}{|H(u, v)|^2 + S_n(u, v)/S_f(u, v)} \right] G(u, v)
\]

Where

\[
\hat{F}(u, v) = \text{Recovered Signal}, H(u, v) = \text{Degrading function}
\]

\[
G(u, v) = \text{Degraded image}
\]

\[
S_n(u, v) = \text{Noise PSD}, S_f(u, v) = \text{Uncorrupted signal PSD}
\]

It may be noted that

\[
S_n(u, v)/S_f(u, v) = 1/\text{SNR}(u, v)
\]

In a noisy environment, in regions where the signal energy dominates the noise energy, the Wiener filter behaves as the inverse filter.

\[
\text{SNR}(u, v) \approx \infty
\]

\[
\hat{F}(u, v) = \left[ \frac{1}{H(u, v)} \frac{|H(u, v)|^2}{|H(u, v)|^2 + 1/\infty} \right] G(u, v)
\]

\[
\hat{F}(u, v) = \left[ \frac{1}{H(u, v)} \frac{|H(u, v)|^2}{|H(u, v)|^2 + 0} \right] G(u, v)
\]

\[
\hat{F}(u, v) = \left[ \frac{1}{H(u, v)} \right] G(u, v)
\]
In regions where the noise energy dominates, the Wiener filter attenuates, suppressing the undesirable noise.

\[ SNR(u, v) \approx 0 \]

\[ \hat{F}(u, v) = \left[ \frac{1}{H(u, v)} \frac{|H(u, v)|^2}{|H(u, v)|^2 + 1/0} \right] G(u, v) \]

\[ \hat{F}(u, v) = \left[ \frac{1}{H(u, v)} \frac{|H(u, v)|^2}{|H(u, v)|^2 + \infty} \right] G(u, v) \]

\[ \hat{F}(u, v) = 0 \]

In the context of Labeyrie’s deconvolution, this model for a noise corrupted signal doesn’t apply perfectly. This is because rather than deconvolving a degrading image from a single degraded image, we are attempting to deconvolve an average modulus squared degrading image from an average modulus square degraded image. This means that the Wiener filter will not perform exactly as it was designed to do, but the basic concepts of inverse filtering in areas of high SNR and attenuating in areas of low SNR still apply. To apply the Wiener filter to Labeyrie’s deconvolution, its variables are set as:

\[ \hat{F}(u, v) = \hat{O}(u, v), H(u, v) = \langle |P(u, v)|^2 \rangle \]

\[ G(u, v) = \langle |I(u, v)|^2 \rangle \]

\[ S_n(u, v) = \langle |N(u, v)|^2 \rangle^2, S_f(u, v) = \langle |O(u, v)|^2 \rangle^2 \]

So the Wiener filter becomes

\[ \langle |\hat{O}(u, v)|^2 \rangle = \left[ \frac{\langle |P(u, v)|^2 \rangle^2}{\langle |P(u, v)|^2 \rangle^2 + \langle |N(u, v)|^2 \rangle^2 / \langle |O(u, v)|^2 \rangle^2} \right] \langle |I(u, v)|^2 \rangle \]
Using the ideal Wiener filter requires full knowledge of the PSD of the noise as well as the uncorrupted signal. For the presented model including additive gaussian white noise and shot noise, the distribution of noise energy is uniform over spatial frequency spectrum [30]. This means that a constant may be used for the noise energy variable \( N(u, v) \). Because the PSD of the uncorrupted signal is often unknown, a common simplification is to use a constant value for the signal PSD \( O(u, v) \) as well [23], resulting in the following simplification of the Wiener filter:

\[
< |\hat{O}(u, v)|^2 > = \left[ \frac{< |P(u, v)|^2 >^2}{< |P(u, v)|^2 >^2 + K} \right] < |I(u, v)|^2 > < |P(u, v)|^2 >
\]

As \( K \) is increased, more of the central lobe is attenuated, along with increased attenuation of noise. An excessively high \( K \) value will attenuate high frequency components of the desired signal, which causes the autocorrelation’s lobes to be wider, preventing resolving of closely spaced binaries. A value of \( K \) is chosen experimentally to give the greatest contrast between the desired, diffraction limited information and the background noise while still giving a sharp autocorrelation image. The high signal energy regions where inverse filtering is performed is shown in Figure 3.6. The results of this simplified Wiener filtering is shown in Figure 3.7.

For this simulated binary and reference star data, this simplified Wiener filter very effectively removes the atmospheric distortion frequency components while keeping wideband noise attenuated to reasonable levels, recovering a close to ideal diffraction limited image of the binary star.

When processing data from KPNO, additional high frequency noise was observed in calculated PSDs, potentially from electrical interference. This noise was
Figure 3.6: Wiener filter inverse filter region

Figure 3.7: Results from simplified Wiener filter deconvolution
not present in simulated data, and so extra measures were taken to deal with this noise. Example data of binary and reference star images along with their average PSDs is shown in Figure 3.8.

![Binary Star Image](image1)
![Reference Star Image](image2)

![Binary Star PSD](image3)
![Reference Star PSD](image4)

**Figure 3.8: Binary and Reference star data from KPNO**

Noise is visible along the axes of the reference star PSD. When deconvolving with the simplified Wiener filter discussed previously, this noise remains visible in the deconvolved PSD, seen in Figure 3.9. Trying to adjust the constant $K$ variable
to improve the quality of the deconvolution is ineffective, increasing $K$ attenuates noise but also the signal. Setting $K$ for sharp lobes leaves high frequency noise in the image, seen in the background of the image.

![Deconvolved PSD](image1.png) ![Deconvolved Acorr](image2.png)

**Figure 3.9: Applying simplified Wiener filter to KPNO data**

To minimize this high frequency noise, knowledge of the signal to be recovered is employed. Because both the binary and reference star images are captured through the telescope, it is known that the image captured by the camera will be confined to the spatial frequency band passed by the telescope. So the $S_f(u,v)$ term, the PSD of the recovered signal, may be set to a lowpass function instead of a constant. This has the effect of emphasizing the low frequency information and attenuating the high frequency information, which is known to be noise. To prevent ringing in the corresponding autocorrelation, a Gaussian lowpass filter is used for $S_f(u,v)$ with a cutoff frequency slightly higher than the telescope’s cutoff frequency. The constant $S_n(u,v)$ term remains a constant, and is adjusted to give
satisfactory deconvolution. Results of this Wiener filter deconvolution are shown in Figure 3.10.

![Figure 3.10: Applying second Wiener filter to KPNO data](image)

This form of the Wiener filter allows greater control over the region to be filtered vs attenuated. It may be seen that the background noise present in 3.9 is no longer visible in 3.10. A comparison of the regions where inverse filtering is performed (regions of high SNR) with these two Wiener filtering methods is shown in Figure 3.11. Wiener filter #1 refers to the first Wiener filter implemented with a constant SNR, Wiener filter #2 refers to the second Wiener filter implemented with a constant $S_n(u, v)$ and Gaussian lowpass $S_f(u, v)$. Ideally, the Wiener filter would perform inverse filtering out to the diffraction limited spatial frequency of the telescope’s optics (where the SNR is good), and not perform it outside this region (where the SNR is poor). Figure 3.11 shows that the Wiener filter #2 inverse filters up to nearly the same frequencies as Wiener filter #1, yet attenuates more high frequency noise than Wiener filter #1. Wiener filter #2, with its more accurate $S_f(u, v)$
model, gives a recovered autocorrelation with more visible and sharper side lobes, allowing for easier extraction of astrometric data.

![Wiener Filter #1](image1.png) ![Wiener Filter #2](image2.png)

**Figure 3.11: Comparison of inverse filtering regions of Wiener filters**

One downside of this Wiener filtering method is that it requires different values of \( S_n(u, v) \) for different images, depending on the relative brightness of the binary and reference stars as well as other factors. One simple alternative that works well consistently between different binary/reference star pairs is just using a gaussian lowpass filter to attenuate the high frequency noise in the inverse filtered PSD (this was the approach used in the PS3 program). This gives fairly similar results to the Wiener filter often gives, as it also preserves the low frequencies high in SNR and attenuates the high frequencies with low SNRs. This is implemented in the Labeyrie Deconvolution modules in Appendix F. This results of this deconvolution method are shown below in Figure 3.12.
Labeyrie’s speckle interferometry process allows for the calculation of astrometric data of binary stars in the presence of atmospheric distortion. Using a reference star allows for calculating astrometric data in poor seeing conditions.

Python scripts used to generate the images in this section are found in Appendix E.

3.3 Implementation

To allow for automation of speckle interferometry by students and instructors at different universities, this Labeyrie deconvolution method was implemented in several Python modules. Example code was written implementing these modules to preprocess large quantities of .FITS data cubes to PSD .FITS files and creating autocorrelations from deconvolved binary/reference star PSDs. The 1000-512x512 image .FITS data cubes that have been commonly used for observations take up 1GB when uncompressed. This means that from several nights of observing, data
on the order of 500GB-1TB can be generated. This makes it difficult to transfer the
data over the internet, which is especially important for remote observatories. The
preprocessed PSD .FITS file from a 1GB .FITS data cube is 1.1MB, and holds all
the information needed for this Labeyrie Deconvolution process. This is a nearly
1000x reduction in data size, making data transfer significantly faster. The modules
written are shown in Appendix F and the scripts that use them to preprocess and
deconvolve are shown in Appendix G.
4. IMPLEMENTATION IN AUTOMATED OBSERVATORY

4.1 Optimization

The present tool used for processing speckle interferometry data by Dr. Genet's students is the "PS3" tool, written by David Rowe [31]. This is a fully featured tool for deconvolving reference from binary stars, preprocessing data, filtering data, and other functions. The basic workflow and timing for using PS3 to generate the autocorrelation of a deconvolved binary and reference star is shown below in Figure 4.1. The workflow revolves around working with large .FITS cubes of data from a past observation.

![Figure 4.1: Workflow with PS3 Software](image)

This shows that the vast majority of the time is spent in the preprocessing stage, where the PSD of each image is calculated and averaged. Common data set sizes are around 1000 images, so speeding up the calculation of the PSD could greatly increase the speed of the overall calculation.
Various methods were used to calculate the PSD of each image in Python to find the fastest. The most basic method was working only in Python, utilizing the NumPy and SciPy FFT algorithms. In an attempt to further increase calculation speed, the PSD calculation was done in Cython, with the expectation that the static type declarations could speed up these calculations. Finally, the FFTW FFT algorithm was used in a PSD calculating C function, which was then wrapped for use in Python using the Python ctypes module. The computation time of the PSD of a 512x512 float32 image using each technique is shown below in Figure 4.2, with the code used for this benchmark shown in Appendix H:

![Figure 4.2: Comparison of PSD Calculation Techniques](image-url)
The wrapped C code, which utilizes the notoriously fast FFTW algorithm, was the fastest. One of the large gains from using the FFTW function is its special "R2C" function, for calculating the DFT of real data. The DFT of real data will always be symmetric and therefore contain redundant data [32]. The FFTW R2C DFT only calculates the non-redundant data, speeding up the calculation. The most visible tradeoff for this speed increase was simply development time: NumPy/SciPy solutions required practically zero extra research effort while utilizing wrapped C code required some effort to pass NumPy arrays back and forth between Python and C code. Using this fast PSD calculation inside the preprocessing script written in Python (seen in Appendix G) results in a preprocessing calculation of 10 seconds. Compared to the average 120 seconds required to preprocess in the PS3 program, this is a 12x improvement.

While this speed increase is useful, greater gains for the user may be realized by implementing these PSD calculations earlier in the observation process.

4.2 Implementation

The common workflow that has been used for observing visual binary stars using speckle interferometry is shown below in Figure 4.3:

Because of the many images being taken, the observation time is a significant portion of the overall time. Presently, during the observation time the only task in progress is the capturing of photos by the camera. This means that the computer used for the observation is sitting idly, not being fully utilized. With more au-
tomation, the observation and preprocessing time may be combined to give faster results.

Dr. Ridgely is currently designing an assembly of flip mirrors, filter wheels, cameras, and a computer to be mounted on a telescope. This assembly would allow for several different observations without changing equipment, would be suitably rigid/integrated to give repeatable results, and would only require connection of one cable each for power and communication (versus the array of communication/power cables that would be required if each part was controlled separately). In addition, software is being designed to automate all of these parts together. A large part of this software is improving the user experience of capturing speckle interferometry data. One of the major improvements to be made is the display of an autocorrelation in near real time, which would allow the observer to confirm that the target and settings were correct, preventing time-wasting errors. For this to be possible, the proposed workflow shown below in Figure 4.4 is to be implemented:

Because the wrapped C function allows for 9ms PSD calculations and the exposure and data transfer overhead time of an individual image is often in the range of 10-100ms, the PSD of each image may be calculated and accumulated immediately after it is received by the computer by starting a thread in parallel to the main picture taking thread. This results in a near real-time average PSD available to the user, which may be transformed into an autocorrelation whenever desired. This means that the deconvolved autocorrelation of a reference and binary star may be displayed almost immediately after both observations have been completed, as there is no more preprocessing task to be done.
Utilizing this automated camera assembly and software should allow for fewer mistakes and less time wasted, allowing for more high-quality data to be taken during an observing run at a large telescope.
Figure 4.3: Present workflow for binary star observations

1. Slew to Binary Star 5-20s
2. Capture Image 0.01-0.1s
3. 1000 Images Captured? N Y
   - 1000 Images Captured? N Y
4. Slew to Reference Star 5-20s
5. Capture Image 0.01-0.1s
6. 1000 Images Captured? N Y
7. Preprocess Binary Star Data 120s
8. Preprocess Reference Star Data 120s
9. Calculate Autocorrelation 1s
Figure 4.4: Proposed workflow for binary star observations
5. FUTURE WORK

The implementation of these Python modules in automated observatories is the next step after this thesis. As previously explained, these modules will be used in John Ridgely’s camera assembly to allow for near real-time viewing of PSDs of observed stars and reduce the workload of the astronomer.

The next major step in this project is the implementation of tools for doing phase recovery, which would allow for recovery of the original images with no ambiguity. Recovery of original images allows for taking photometric measurements as well. All of the phase recovery techniques are very computationally intense, and could be sped up significantly by utilizing a coprocessor, such as the Intel Xeon Phi.

All the simulations in this thesis are more qualitative than quantitative, and could be improved by students stronger in the fields of image processing and computer science. This would allow for superior simulations of real observed conditions (with a measured Fried parameter or specified noise value), rather than a more qualitative demonstration of a phenomenon.
6. CONCLUSIONS

Observation of close visual binary stars is limited by both the telescope used for observation as well as the atmosphere. A simulation of the effects of telescope optics and atmospheric distortion as well as shot and additive noise was created for generating simulated binary star data. The deconvolution process originally documented by Labeyrie [27] allows for recovery of the magnitude information from speckle images of binary stars. Simulated data as well as real data taken at Kitt Peak National Observatory was successfully processed by Python modules for performing Labeyrie’s deconvolution process, giving diffraction limited autocorrelations. A Wiener filter with a constant SNR was shown to be effective for deconvolving simulated data. Additional noise sources in real data warranted the use of a Wiener filter modeling the passband of the telescope’s aperture to attenuate noise and recover signal data out to the diffraction limit. For processing large sets of real data with no modification to the filter between binary stars, a lowpass filtered inverse filter was found to give more consistent results than either Wiener filter. The calculation of the power spectral density was shown to be the most computationally intense part of this process, and was optimized from 25ms to 9ms to allow for near real time PSD calculation while observing.

All code generated during this project may be found at the following GitHub repository: https://github.com/nsmidt/CPSLO-Speckle-Reduction.
REFERENCES


A. Prime Focus Geometric Optics

Simulation Simulation of parabolic lens focusing light of a plane wave

**Kitt Peak 2.1m Telescope**

From [http://www-kpno.kpno.noao.edu/kpno-misc/2m_params.html](http://www-kpno.kpno.noao.edu/kpno-misc/2m_params.html), this telescope has the following specs:

- Diameter = 2.133m = 2133mm
- Radius of curvature = 11168.4mm
- Effective Focal Length = R/2 = 5584mm
- Focal Ratio = 5584/2133 = f/2.63

```
sim_prime_focus.py

import matplotlib.pyplot as plt
import numpy as np

# Lens Parameters [mm]
diameter = 2133
focal_length = 5584

# Equation for reflector’s shape
def parabola(x):
    return (x**2)/(4*focal_length)

# Number of rays to be plotted
num_rays = 5

# Incident angles to be plotted
angles = [-5, -2.5, 0, 2.5, 5]

# Setting up Figure
plt.figure(figsize=(6,11))

# Loop through all incident angles
for i, angle in enumerate(angles):
    # Create a new subplot
```
plt.subplot(len(angles),1,i+1)

# Incident ray angle
angle_incident = np.deg2rad(angle) # In radians

# Reflection Locations
# Calculating reflection locations
reflection_y = np.zeros(num_rays)
for i in np.arange(num_rays):
    # Evenly space rays over the diameter of reflector
    reflection_y[i] = -diameter/2 + i*diameter/(num_rays - 1)
# Calculating mirror displacement at reflection location
reflection_x = -parabola(reflection_y)

# Start Locations
# Setting X value where incident rays come from
start_x = -focal_length*1.2
# Calculating Y values where incident rays come from
# Simple Y = mx+b calculation
start_y = reflection_y+(start_x-reflection_x)*np.tan(angle_incident)

# Calculating reflected angles of
# each ray using ray trace matrix eqn
angle_reflected = angle_incident - (1/focal_length)*
    reflection_y

# End locations
# Calculating X values where reflected waves end
end_x = -focal_length
# Calculating Y values where incident rays come from
# Simple Y = mx+b calculation
end_y = reflection_y+(reflection_x-end_x)*np.tan(angle_reflected)

# Draw Subplot

# Plot parabolic lens
y = np.arange(-diameter/2,diameter/2,diameter/1000)
x = -parabola(y)
plt.plot(x,y,linewidth=5.0)

# Plot incident light rays
for i in np.arange(num_rays):
    line = plt.Line2D((start_x,reflection_x[i]), (start_y[i],reflection_y[i]), lw=1)
plt.gca().add_line(line)

# Plot reflected light rays
for i in np.arange(num_rays):
    line = plt.Line2D((end_x, reflection_x[i]), (end_y[i], reflection_y[i]), lw=1)
    plt.gca().add_line(line)

plt.axis("equal")
plt.grid("on")

plt.show()
B. Prime Focus Diffraction Limit Simulation

Simulating the output of a prime focus telescope given an input image.

**Kitt Peak 2.1m Telescope**

From [http://www-kpno.kpno.noao.edu/kpno-misc/2m_params.html](http://www-kpno.kpno.noao.edu/kpno-misc/2m_params.html), this telescope has the following specs:

- Diameter = 2133.0mm
- Radius of curvature = 11168.4mm
- Effective Focal Length = R/2 = 5584mm
- Focal Ratio = 5584/2133 = f/2.63

From [http://www.jdso.org/volume11/number1s/Genet_234_244.pdf](http://www.jdso.org/volume11/number1s/Genet_234_244.pdf), telescope was set up differently for speckle observation. Adding secondary mirror changes focal length:

- Primary Diameter = 2.1m = 2133.0mm
- Focal Ratio = f/7.6
- Focal Length = 2133.0mm*7.6 = 16211mm
- 8x Barlow
- Focal Ratio = f/(7.6*8) = f/60.8
- Focal Length = 16211mm*8 = 129.69m

**Andor Luca-R**

From [https://www.andor.com/pdfs/specifications/Andor_Luca-R_604_Specifications.pdf](https://www.andor.com/pdfs/specifications/Andor_Luca-R_604_Specifications.pdf)

- Sensor Size = 1004 (H) x 1002 (V) Pixels = 8mm X 8mm
- Pixel Size = 8um X 8um
- Only center 512x512 pixels used

**Calcs:**

Input image is in units of angles, represents the full input field of view

- This is re-sampled to image it on a pixel scale

- image scale \(= \frac{206265}{focal\ length} = \frac{206265\ arcsec}{129.69 \times 10^3} = 1.59\ arcsec/mm\)

- pixel scale \(= 1.59 \frac{as}{mm} \times 8 \times 10^{-3} \frac{mm}{pixel} = 12.72\ arcsec/pixel\)

- If sky input is in units of pixel scale [ex: 12.72 marcsec], then no re-sampling required

Aperture is in length units [m]

- Need to calculate the "effective" size of the aperture, taking into account effects of lens focal length and light wavelength:

- Effective Aperture Diameter \(= \frac{Aperture\ Diameter}{(\lambda)/(Focal\ Length)}\)

- \(D = \frac{(2.133m)}{(0.8 \times 10^{-6}m)(129.69 m)} = 20599 \frac{1}{m}\)

- This is the radius of a circle that gives the first null of the corresponding airy disc to be at \(r = \frac{1.22}{D} = \frac{1.22}{20599 \frac{1}{m}} = 59.34 \times 10^{-6} m\)

- In pixels, this first null location corresponds to \(\frac{59.34 \times 10^{-6} m}{8 \times 10^{-6}} = 7.42\ pixels\)

To simulate the imaging sensor, we need the sample units of the FFT of our aperture image to be the size of a pixel. For Andora Luca-R, pixel is 8um. Each sample of a 2D FFT represents \(2/L\) (\(L = \) total sample size of input image). So we want total spatial sampling range \(= \frac{1}{L} = 8 \times 10^{-6} \rightarrow L = \frac{1}{8 \times 10^{-6}} = 125 \times 10^3\)

- If this is sampled with 512 pixels, each pixel represents \(\frac{125 \times 10^{-3}}{512} = 244.14 \frac{1}{m}\). So
the diameter of the aperture \( D_s = \frac{20599}{244.14} = 84 \text{ pixels} \)

- The FFT of this is calculated, PSF is calculated as square of magnitude

- Sanity check: first null occurs at 7 pixels away from center (256)

- Input and aperture PSF are convolved

```
sim_diffraction_limit.py

# Includes
import matplotlib.pyplot as plt
import numpy as np
from classes_atmos_sim import atmospheric_simulation
from scipy.signal import argrelextrema

# Size of simulated image
nxy = 512

# Instantiate simulation object
sim = atmospheric_simulation()

## Assign all variables of simulation
# Aperture/Telescope Specifications:
sim.diameter_m = 2.133 # Mirror diameter in meters
sim.focal_length = 129.69 # Effective focal length in meters

# Camera Specs
sim.wavelength = 0.8E-6 # Wavelength of light
sim.pixel = 8E-6 # length of pixel side in m
sim.bits = 14 # Bits in camera ADC
sim.nxy = nxy # Length of sensor side in pixels
sim.center = int(nxy/2) # Center of sensor in pixels
sim.platescale = 206265*sim.pixel/(sim.focal_length) #
    Calculate plate scale
sim.gamma = 1.6 # Gamma correction

# Binary star specs
sim.rho = 0.5 # Set separation in arcseconds
sim.phi = 0 # Set angle in degrees

# Initializing Empty Images
sim.input_img = np.zeros((nxy,nxy)) # Input binary star object
sim.aperture_screen_s = np.zeros((nxy,nxy)) # Aperture screen
sim.pupil_screen = np.zeros((nxy,nxy)) # Total Pupil Screen
```
```python
# Atmosphere
sim.atmosphere_screen = np.zeros((nxy,nxy))
# PSF of aperture
sim.psf = np.zeros((nxy,nxy))
# Simulated Binary Image
sim.binary_img = np.zeros((nxy,nxy))

## Run Simulation
# Create ideal binary star image
sim.create_input_image()
# Create telescope aperture image
sim.create_aperture()
# Create atmospheric distortion phase screen
# We aren't simulating atmospheric distortion, so we just set
# this to a constant
sim.atmosphere_screen.fill(1)
# Calculate PSF of aperture and atmospheric phase screen
sim.get_psf()
# Calculate simulated binary star image from PSF
sim.get_binary()

# Incices for zoomed view
dx = 100 # Number of points to be plotted around the center of
        # image
x = np.arange(int(sim.center-dx/2),int(sim.center+dx/2))
fov = sim.center*sim.platescale

# Plots
colormap = "gray"
plt.figure(figsize = (14,6), dpi = 100)
plt.subplot(1,2,1)
plt.imshow(sim.aperture_screen_s, cmap=colormap, extent = (-
        sim.X_aperture_s_meff/2,sim.X_aperture_s_meff/2,-sim.
        X_aperture_s_meff/2,sim.X_aperture_s_meff/2))
plt.xlabel("[m]")
plt.ylabel("[m]")
plt.title("Aperture Screen")
plt.subplot(1,2,2)
plt.imshow(np.log10(sim.psf), cmap=colormap)
plt.xlabel("[pixels]")
plt.ylabel("[pixels]")
plt.title("Aperture PSF")
plt.figure(figsize = (14,6), dpi = 100)
plt.subplot(1,2,1)
plt.imshow(sim.input_img, cmap=colormap)
```

plt.axis([x[0],x[dx-1],x[0],x[dx-1]])
plt.title("Zoomed Input Image")
plt.subplot(1,2,2)
plt.imshow(sim.binary_img, cmap=colormap)
plt.axis([x[0],x[dx-1],x[0],x[dx-1]])
plt.title("Zoomed Sensor Image")

# Show center row of aperture PSF and sensor’s image
plt.figure(figsize = (20,10), dpi = 100)
# Aperture PSF Plot
plt.subplot(1,2,1)
plt.plot(x,sim.psf[sim.center,x])
plt.title("Aperture PSF (Center Row)")

# Output Image Plot
plt.subplot(1,2,2)
plt.plot(x,sim.binary_img[sim.center,x])
plt.title("Output Image (Center Row)")

# Show figures
plt.show()

# List distances from center of nulls in PSF
print("Aperture PSF Null Radii")
minima_i = argrelextrema(sim.psf[sim.center], np.less)
minima_i = np.subtract(minima_i,sim.center)[0]
print_num = 5
minima_found = 0
for value in minima_i:
    if (value > 0) and (minima_found < print_num):
        print(value)
        minima_found = minima_found + 1

Simulation uses Atmospheric Simulation Class found in Appendix D
C. Atmospheric Distortion Simulation

- Atmospheric distortion simulated as a phase component of the pupil plane screen
- Phase component image of pupil screen has the following PSD:

\[
PSD_{phase\_image} = \alpha \cdot k^{-11/3}
\]

\[
phase_{phase\_image} = \text{gaussian with mean=0, variance=1}
\]

These are combined to give the frequency domain representation of the phase image as

\[
phase\_screen\_F = \sqrt{PSD_{phase\_image} \cdot \exp(j \cdot phase_{phase\_image})}
\]

- Take iFFT of frequency domain representation of phase to get phase values

\[
phase\_screen = \text{iFFT}(phase\_screen\_F)
\]

- Create pupil screen as:

\[
pupil\_screen = |aperture\_mask| \cdot \exp(j \cdot phase\_screen)
\]

**sim_atmos_distort.py:**

```python
# Includes
import matplotlib.pyplot as plt
import numpy as np
from classes_atmos_sim import atmospheric_simulation

# Size of simulated image
nxy = 512

# Instantiate simulation object
sim = atmospheric_simulation()

## Assign all variables of simulation
```
# Aperture/Telescope Specifications:
sim.diameter_m = 2.133 # Mirror diameter in meters
sim.focal_length = 129.69 # Effective focal length in meters

# Camera Specs
sim.wavelength = 0.8E-6 # Wavelength of light
sim.pixel = 8E-6 # length of pixel side in m
sim.bits = 14 # Bits in camera ADC
sim.nxy = nxy # Length of sensor side in pixels
sim.center = int(nxy/2) # Center of sensor in pixels
sim.platescale = 206265*sim.pixel/(sim.focal_length) #
   Calculate plate scale
sim.gamma = 1.6 # Gamma correction

# Binary star specs
sim.rho = 0.5 # Set separation in arcseconds
sim.phi = 45 # Set angle in degrees

# Atmospheric Specs
sim.alpha = 1/100 # Multiplicative constant
sim.r0 = 0.2 # Fried Parameter

# Initializing Empty Images
sim.input_img = np.zeros((nxy,nxy)) # Input binary star object
sim.aperture_screen_s = np.zeros((nxy,nxy)) # Aperture screen
sim.phase_screen = np.zeros((nxy,nxy)) # Phase screen values
   (unwrapped)
sim.atmosphere_screen = np.zeros((nxy,nxy)) # Atmosphere screen
sim.pupil_screen = np.zeros((nxy,nxy)) # Total Pupil Screen
sim.psf = np.zeros((nxy,nxy)) # PSF of aperture/atmosphere
sim.binary_img = np.zeros((nxy,nxy)) # Simulated Binary Image

## Run Simulation
# Create ideal binary star image
sim.create_input_image()
# Create telescope aperture image
sim.create_aperture()
# Create atmospheric phase screen image
sim.create_atmosphere_screen()
# Calculate PSF of aperture and atmospheric phase screen
sim.get_psf()
# Calculate simulated binary star image from PSF
sim.get_binary()

## Adding photon shot noise
# Photon numbers to test
photons = (1000,100000)
# Number of tests to run
photons_n = len(photons)
# Shot Noise Images
img_shot_noise = np.zeros((photons_n,nxy,nxy))
# Loop through photon values
for i in np.arange(photons_n):
    # Add Poisson Noise
    img_shot_noise[i] = sim.add_noise(sim.binary_img, photons=photons[i], gaussian_var=0)

## Comparing effects of shot, gaussian, and gaussian/shot noise
# Gaussian noise variance (in photons)
var = 3
# Shot noise number of received photons
photons2 = 50E3
# Noise Images
img_shot_noise2 = np.zeros((nxy,nxy))
img_gaussian_noise = np.zeros((nxy,nxy))
img_gaussian_shot_noise = np.zeros((nxy,nxy))
# Calculate Noise Images
binary_img_scaled = photons2*(np.abs(sim.binary_img)/np.sum(sim.binary_img))
img_shot_noise2 = sim.add_noise(binary_img_scaled, photons=photons2, gaussian_var=0)
img_gaussian_shot_noise = sim.add_noise(binary_img_scaled, photons=photons2, gaussian_var=var)

## Plots
colormap = "jet"

plt.figure(figsize = (18,8))
plt.subplot(1,2,1)
plt.imshow(sim.phase_screen, cmap=colormap)
plt.colorbar()
plt.title("Complex Pupil Phase")
plt.subplot(1,2,2)
plt.imshow(sim.aperture_screen_s, cmap=colormap)
plt.title("Complex Pupil Magnitude")
plt.figure(figsize = (8,8))
plt.imshow(sim.psf, cmap=colormap)
plt.title("Total System PSF")
Simulation uses Atmospheric Simulation Class found in Appendix D.
D. Atmospheric Distortion Simulation Class

classes_atmos_sim.py

# Includes
import matplotlib.pyplot as plt
import numpy as np
from scipy.fftpack import fft2, ifft2, fftshift
from scipy.signal import fftconvolve
from scipy.signal import argrelextrema

## Parameters
# Image specs
nxy = 512
center = int(nxy/2)

class atmospheric_simulation():
    def __init__(self):
        # Aperture/Telescope Specifications:
        self.diameter_m = None  # Mirror diameter in meters
        self.focal_length = None  # Effective focal length in meters

        # Camera Specs
        self.wavelength = None  # Wavelength of light
        self.pixel = None  # length of pixel side in m
        self.bits = None  # Bits in camera ADC
        self.nxy = None  # Length of sensor side in pixels
        self.center = None  # Center of sensor in pixels
        self.platescale = None  # Plate scale
        self.gamma = None  # Gamma correction

        # Binary star specs
        self.rho = None  # Set separation in arcseconds
        self.phi = None  # Set angle in degrees

        # Atmospheric Specs
        self.alpha = None  # Multiplicative constant
        self.r0 = None  # Fried Parameter

        # NOTE: R0 DOESN’T YET PROPERLY MODEL A REAL R0 VALUE,
        # ALPHA MUST BE
        # PROPERLY SET TO GIVE CORRECT COMPARISON TO REAL R0
        # VALUES. DID NOT
        # HAVE TIME TO CALCULATE THIS VALUE
# Initializing Empty Images
self.input_img = None  # Input binary star object
self.aperture_screen_s = None  # Aperture screen
self.phase_screen = None  # Phase screen values (unwrapped)
self.atmosphere_screen = None  # Atmosphere screen
self.pupil_screen = None  # Total Pupil Screen
self.psf = None  # PSF of aperture/atmosphere
self.binary_img = None  # Simulated Binary Image

def create_input_image(self):
    # Calculate coordinates of stars
    x = int(self.rho/(2*self.platescale) * np.cos(np.deg2rad(self.phi)))
    y = int(self.rho/(2*self.platescale) * np.sin(np.deg2rad(self.phi)))
    x1 = self.center + x
    y1 = self.center + y
    x2 = self.center - x
    y2 = self.center - y

    # Place stars on image
    self.input_img[y1,x1] = 1
    self.input_img[y2,x2] = 1
    # Scale image power to 1
    input_img_power = np.sum(np.power(self.input_img,2))
    self.input_img = np.divide(self.input_img,np.sqrt(input_img_power))

def emphasized_image(self, img, circle_radius = 3):
    # Convolve small circles with input images to make them more visible
    # Create meshgrid
    xx,yy = np.meshgrid(np.arange(self.nxy),np.arange(self.nxy))
    # Calculate grid of distances from circle center
    radius = (xx-center)**2 + (yy-center)**2
    # Draw boolean circle
    circle = (radius < (circle_radius**2)).astype(np.int64)
    # Convolve circle with difficult to see images
    img_emph = fftconvolve(img,circle)[self.center:self.center+1+np.arange(self.nxy)]
    # Return emphasized image
    return img_emph
def create_aperture(self):
    ## Telescope aperture creation:
    # Total spatial sample range
    X_aperture_s = 1/self.pixel
    # dx value for sampled aperture image
    dx_aperture_s = X_aperture_s/nxy
    # Coordinates of sampled image
    x_aperture_s = np.arange(0,X_aperture_s,dx_aperture_s) - X_aperture_s/2
    # Meshgrid of sampled coordinates
    xx_s,yy_s = np.meshgrid(x_aperture_s,x_aperture_s)
    # Scaled aperture diameter to effectively resample
    diameter_s = self.diameter_m/(self.focal_length*self.wavelength)
    # Draw new circle at correct dimensions
    # Calculate grid of distances from circle center
    circle_s = (xx_s)**2+(yy_s)**2
    # Draw boolean circle
    circle_s = circle_s<(diameter_s/2)**2
    # Convert boolean circle to int
    circle_s = circle_s.astype(np.int64)

    # Scale aperture image power to 1
    aperture_screen_power = np.sum(np.power(circle_s,2))
    # Save aperture image in units of meters
    self.aperture_screen_s = np.divide(circle_s,np.sqrt(aperture_screen_power))

    # Calculate effective size of sampled aperture image in meters
    self.X_aperture_s_meff = self.focal_length*self.wavelength/self.pixel

def create_atmosphere_screen(self):
    ## Phase screen creation:
    # Generate random image
    # To be used in creating random atmospheric element
    phase_phase = np.multiply(np.pi,np.random.normal(loc=0,
        scale=1,size=(nxy,nxy)))
    # Total array sample size
    d_aperture = self.focal_length*self.wavelength/self.pixel
    # Spatial sample resolution
    dxy = d_aperture/nxy
    # Spatial frequency resolution
    df = 1/(d_aperture)
    # Image sample indices array
x = np.multiply( np.subtract(np.arange(self.nxy),self.center), dxy )
# Spatial Frequency indices array
xf = np.multiply( np.subtract(np.arange(self.nxy),self.center), df )
# Meshgrid of spatial frequency domain
[xx,yy]=np.meshgrid(xf,xf)
# Radius from center meshgrid
rr = (np.sqrt(np.power(xx,2)+np.power(yy,2)))
# Calculate Kolmogorov spectral density
phase_PSD = np.power(rr, -11/3)
phase_PSD = np.multiply(self.alpha*0.023/(self.r0**(5/3)), phase_PSD)
# Set DC component to 0 (previous calc attempts to set to 1/0)
phase_PSD[self.center,self.center] = 0
# Construct phase screen spectrum
phase_screen_f = np.multiply(np.sqrt(phase_PSD),np.exp(1j*phase_phase))
# Calculate phase screen
self.phase_screen = np.real(ifft2(fftshift(phase_screen_f)*nxy*nxy))
# Create complex atmospheric screen
self.atmosphere_screen = np.exp(np.multiply(1j,self.phase_screen))

def get_psf(self):
    # Generate total screen, combining atmosphere and aperture
    self.pupil_screen = np.multiply(self.atmosphere_screen, self.aperture_screen_s)

    ## Calculate system’s total response
    # Calculate total PSF of system
    self.psf = fftshift(fft2(self.pupil_screen))
    self.psf = np.power(np.abs(self.psf),2)
    # Normalize PSF
    psf_power = np.sum(np.power(self.psf,2))
    self.psf = np.divide(self.psf,np.sqrt(psf_power))
    # Gamma correct PSF
    self.psf = np.power(self.psf,1/self.gamma)
    # Normalize PSF
    psf_power = np.sum(np.power(self.psf,2))
    self.psf = np.divide(self.psf,np.sqrt(psf_power))

def get_binary(self):
    # Convolve PSF with input image using FFT
    self.binary_img = fftconvolve(self.input_img, self.psf)
# Save the center 512x512 image

# Normalize binary
binary_power = np.sum(np.power(self.binary_img, 2))
self.binary_img = np.divide(self.binary_img, np.sqrt(binary_power))

def add_noise(self, img, photons, gaussian_var):
    # Array to be returned
    img_noisy = np.array(img)

    # If photons > 0, add shot noise
    if (photons > 0):
        # Normalize image to number of photons incident
        img_noisy = photons * (np.abs(img) / np.sum(img))
        # Calculate image with shot noise
        img_noisy = np.random.poisson(lam=img_noisy, size=None)

    # If gaussian_var > 0, add additive noise
    if (gaussian_var > 0):
        # Create noise image
        noise = np.random.normal(loc=0, scale=gaussian_var, size=(self.nxy, self.nxy))
        # Add noise image to simulated image
        img_noisy = img_noisy + noise
        # Turn all negative pixels to 0
        img_noisy[img_noisy < 0] = 0

    # Return noisy image
    return img_noisy

# Normalize image between 0 and a given max
def normalize(img, norm_max):
    img_new = np.array(img)
    img_new -= img.min()
    img_new -= img.min()
    img_new = img_new * norm_max / img_new.max()
    return img_new
E. Labeyrie Deconvolution Simulations

Simulation uses Atmospheric Simulation Class found in appendix D.

Simulation of noiseless Labeyrie deconvolution:

`sim_labeyrie_decon.py`

```python
# Includes
import matplotlib.pyplot as plt
import numpy as np
from scipy.fftpack import fft2, ifft2, fftshift
from scipy.signal import fftconvolve
from scipy.signal import argrelextrema
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm
from classes_atmos_sim import atmospheric_simulation

## Parameters
# Image specs
nxy = 512
center = int(nxy/2)
n_exposures = 100 # Number of simulated exposures

# Instantiate simulation object
sim = atmospheric_simulation()

## Assign all variables of simulation
# Aperture/Telescope Specifications:
sim.diameter_m = 2.133 # Mirror diameter in meters
sim.focal_length = 129.69 # Effective focal length in meters

# Camera Specs
sim.wavelength = 0.8E-6 # Wavelength of light
sim.pixel = 8E-6 # length of pixel side in m
sim.bits = 14 # Bits in camera ADC
sim.nxy = nxy # Length of sensor side in pixels
sim.center = int(nxy/2) # Center of sensor in pixels
sim.platescale = 206265*sim.pixel/(sim.focal_length) #
            Calculate plate scale
sim.gamma = 1.6 # Gamma correction

# Binary star specs
sim.rho = 0.5 # Set separation in arcseconds
sim.phi = 45 # Set angle in degrees
```
# Atmospheric Specs
sim.alpha = 1/100 # Multiplicative constant
sim.r0 = 0.2 # Fried Parameter

# Initializing Empty Images
sim.input_img = np.zeros((nxy,nxy)) # Input binary star object
sim.aperture_screen_s = np.zeros((nxy,nxy)) # Aperture screen
sim.phase_screen = np.zeros((nxy,nxy)) # Phase screen values
  (unwrapped)
sim.atmosphere_screen = np.zeros((nxy,nxy)) # Atmosphere screen
sim.pupil_screen = np.zeros((nxy,nxy)) # Total Pupil Screen
sim.psf = np.zeros((nxy,nxy)) # PSF of aperture/atmosphere
sim.binary_img = np.zeros((nxy,nxy)) # Simulated Binary Image

## Run Simulation
# Create image accumulation arrays
psf_avg = np.zeros((nxy,nxy))
binary_psd_avg = np.zeros((nxy,nxy))
reference_psd_avg = np.zeros((nxy,nxy))

# Create ideal binary star image
sim.create_input_image()
# Create telescope aperture image
sim.create_aperture()
# Generate multiple images, integrate them
for i in np.arange(n_exposures):
  # Create atmospheric phase screen image
  sim.create_atmosphere_screen()
  # Calculate PSF of aperture and atmospheric phase screen
  sim.get_psf()
  # Calculate simulated binary star image from PSF
  sim.get_binary()

  # Integrate PSFs
  psf_avg += sim.psf

  # Calculate PSD of binary star image
  binary_psd = np.power(np.abs(fftshift(fft2(sim.binary_img))),2)
  # Integrate PSDs
  binary_psd_avg += binary_psd

  # Calculate PSD of PSF
  reference_psd = np.power(np.abs(fftshift(fft2(sim.psf))),2)
  # Integrate PSDs
  reference_psd_avg += reference_psd
# Calculate average of PSF/PSDs
psf_avg /= n_exposures
binary_psd_avg /= n_exposures
reference_psd_avg /= n_exposures

# Calculate PSDs
input_img_psd = np.power(np.abs(fftshift(fft2(sim.input_img))), 2)

# Calculate Acorrs
input_img_acorr = np.abs(fftshift(ifft2(input_img_psd)))
reference_acorr_avg = np.abs(fftshift(ifft2(reference_psd_avg)))
binary_acorr_avg = np.abs(fftshift(ifft2(binary_psd_avg)))

# Deconvolve reference from binary
deconvolved_psd_avg = np.divide(binary_psd_avg, reference_psd_avg)
deconvolved_acorr_avg = np.abs(fftshift(ifft2(deconvolved_psd_avg)))

colormap = "jet"

plt.figure(figsize = (18,8), dpi = 150)
plt.subplot(1,2,1)
plt.imshow(sim.psf, cmap=colormap)
plt.title("Short Exposure PSF")
plt.subplot(1,2,2)
plt.imshow(psf_avg, cmap=colormap)
plt.title("Long Exposure PSF")

plt.figure(figsize = (16,16), dpi = 50)
plt.subplot(3,3,1)
plt.imshow(sim.emphasized_image(img=sim.input_img, circle_radius=4), cmap=colormap)
plt.title("Binary Star Object")
plt.subplot(3,3,2)
plt.imshow(sim.psf, cmap=colormap)
plt.title("Atmospheric/Aperture PSF")
plt.subplot(3,3,3)
plt.imshow(sim.binary_img, cmap=colormap)
plt.title("Binary Star Image")
plt.subplot(3,3,4)
plt.imshow(np.log10(input_img_psd), cmap=colormap)
plt.title("Binary Star Object PSD")
plt.subplot(3,3,5)
plt.imshow(np.log10(reference_psd_avg), cmap=colormap)
plt.title("Avg PSF PSD")
plt.subplot(3,3,6)
plt.imshow(np.log10(binary_psd_avg), cmap=colormap)
plt.title("Avg Binary Star Image PSD")
plt.subplot(3,3,7)
plt.imshow(sim.emphasized_image(img=input_img_acorr, circle_radius=4), cmap=colormap)
plt.title("Binary Star Object Autocorrelation")
plt.subplot(3,3,8)
plt.imshow(reference_acorr_avg, cmap=colormap)
plt.title("Avg PSF Autocorrelation")
plt.subplot(3,3,9)
plt.imshow(binary_acorr_avg, cmap=colormap)
plt.title("Avg Binary Star Image Autocorrelation")

fig = plt.figure(figsize = (10,10), dpi = 100)
ax = fig.gca(projection='3d')
[xx,yy] = np.meshgrid(np.arange(nxy), np.arange(nxy))
ax.plot_surface(X=xx, Y=yy,
Z=np.log10(reference_psd_avg),
cmap="jet",
linewidth=0,
antialiased=True)
plt.title("Reference Star PSD")

plt.figure(figsize = (14,14), dpi = 100)
plt.subplot(2,2,1)
plt.imshow(np.log10(binary_psd_avg), cmap=colormap)
plt.title("Avg Binary Star Image PSD")
plt.subplot(2,2,2)
plt.imshow(np.log10(deconvolved_psd_avg), cmap=colormap)
plt.title("Avg Deconvolved PSD")
plt.subplot(2,2,3)
plt.imshow(binary_acorr_avg, cmap=colormap)
plt.title("Avg Binary Star Image Autocorrelation")
plt.subplot(2,2,4)
plt.imshow(sim.emphasized_image(img=deconvolved_acorr_avg, circle_radius=4), cmap=colormap)
plt.title("Avg Deconvolved Autocorrelation")

plt.show()

Simulation of Labeyrie deconvolution in the presence of noise:

sim_wiener_deconv.py:
# Includes
import matplotlib.pyplot as plt
import numpy as np
from scipy.fftpack import fft2, ifft2, fftshift
from scipy.signal import fftconvolve
from scipy.signal import argrelextrema
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm
from classes_atmos_sim import atmospheric_simulation

## Parameters

# Image specs
nx = 512
center = int(nx/2)
n_exposures = 100 # Number of simulated exposures

# Instantiate simulation object
sim = atmospheric_simulation()

## Assign all variables of simulation
# Aperture/Telescope Specifications:
sim.diameter_m = 2.133 # Mirror diameter in meters
sim.focal_length = 129.69 # Effective focal length in meters

# Camera Specs
sim.wavelength = 0.8E-6 # Wavelength of light
sim.pixel = 8E-6 # length of pixel side in m
sim.bits = 14 # Bits in camera ADC
sim.nxy = nx # Length of sensor side in pixels
sim.center = int(nx/2) # Center of sensor in pixels
sim.platescale = 206265*sim.pixel/(sim.focal_length) # Calculate plate scale
sim.gamma = 1.6 # Gamma correction

# Binary star specs
sim.rho = 0.5 # Set separation in arcseconds
sim.phi = 45 # Set angle in degrees

# Atmospheric Specs
sim.alpha = 1/100 # Multiplicative constant
sim.r0 = 0.2 # Fried Parameter

## Initializing Empty Images
sim.input_img = np.zeros((nx,nx)) # Input binary star object
sim.aperture_screen_s = np.zeros((nx,nx)) # Aperture screen
sim.phase_screen = np.zeros((nx,nx)) # Phase screen values
→ (unwrapped)
```python
sim.atmosphere_screen = np.zeros((nxy,nxy))  # Atmosphere
sim.pupil_screen = np.zeros((nxy,nxy))  # Total Pupil Screen
sim.psf = np.zeros((nxy,nxy))  # PSF of aperture/atmosphere
sim.binary_img = np.zeros((nxy,nxy))  # Simulated Binary Image

## Run Simulation
# Create image accumulation arrays
binary_psd_avg = np.zeros((nxy,nxy))
reference_psd_avg = np.zeros((nxy,nxy))

# Create ideal binary star image
sim.create_input_image()
# Create telescope aperture image
sim.create_aperture()
# Generate multiple images, integrate them
for i in np.arange(n_exposures):
    # Create atmospheric phase screen image
    sim.create_atmosphere_screen()
    # Calculate PSF of aperture and atmospheric phase screen
    sim.get_psf()
    # Calculate simulated binary star image from PSF
    sim.get_binary()
    # Add noise to binary and reference stars
    sim.psf = sim.add_noise(sim.psf,1E5,1)
    sim.binary_img = sim.add_noise(sim.binary_img,1E5,1)

    # Calculate PSD of binary star image
    binary_psd = np.power(np.abs(np.fft.fft2(sim.binary_img)),2)
    # Integrate PSDs
    binary_psd_avg += binary_psd

    # Calculate PSD of PSF
    reference_psd = np.power(np.abs(np.fft.fft2(sim.psf)),2)
    # Integrate PSDs
    reference_psd_avg += reference_psd

    # Calculate PSD of input image
    input_psd = np.power(np.abs(np.fft.fft2(sim.input_img)),2)

    # Calculate average of PSF/PSDs
    binary_psd_avg /= n_exposures
    reference_psd_avg /= n_exposures

# Calculate Acorrs
input_acorr = np.abs(np.fft.fft2(input_psd))
```

81
reference_acorr_avg = np.abs(fftshift(ifft2(reference_psd_avg)))

binary_acorr_avg = np.abs(fftshift(ifft2(binary_psd_avg)))

## Assigning values for filter experimentation
H = reference_psd_avg  # Degrading Function = reference star PSD
G = binary_psd_avg  # Degraded Function = binary star PSD
h = np.abs(fftshift(ifft2(H)))  # reference star acorr
g = np.abs(fftshift(ifft2(G)))  # binary star acorr

## Inverse filtering
F_hat_inverse = G/H  # Inverse filtering deconvolution
f_hat_inverse = np.abs(fftshift(ifft2(F_hat_inverse)))

## Simplified Wiener filtering
k = 1E13
F_hat_wiener1 = G*(1/H)*((H**2)/(H**2+k))
f_hat_wiener1 = np.abs(fftshift(ifft2(F_hat_wiener1)))

colormap = "jet"

plt.figure(figsize = (14,18), dpi = 100)
plt.subplot(2,3,1)
plt.imshow(np.log10(H), cmap=colormap)
plt.title("Reference Star Image PSD")
plt.subplot(2,3,2)
plt.imshow(np.log10(G), cmap=colormap)
plt.title("Binary Star Image PSD")
plt.subplot(2,3,3)
plt.imshow(np.log10(F_hat_inverse), cmap=colormap)
plt.title("Deconvolved PSD")
plt.subplot(2,3,4)
plt.imshow(h, cmap=colormap)
plt.title("Reference Star Image Autocorrelation")
plt.subplot(2,3,5)
plt.imshow(g, cmap=colormap)
plt.title("Binary Star Image Autocorrelation")
plt.subplot(2,3,6)
plt.imshow(f_hat_inverse, cmap=colormap)
plt.title("Deconvolved Autocorrelation")

plt.figure(figsize = (6,10), dpi = 100)
plt.subplot(1,2,1)
plt.imshow(np.log10(F_hat_wiener1), cmap=colormap)
plt.title("Deconvolved PSD")
plt.subplot(1,2,2)
plt.imshow(f_hat_wiener1, cmap=colormap)
plt.title("Deconvolved Autocorrelation")

plt.figure(figsize = (10,10), dpi = 100)
plt.imshow((H**2)/(H**2+k), cmap=colormap)
plt.title("Simplified Wiener")
plt.show()

Applying deconvolution to real data from KPNO. Uses Labeyrie Deconvolution class shown in Appendix F:

**sim_kpno_deconv.py:**

```python
import sys, os
from classes_labeyrie import target, deconvolved
import tkinter as tk
from tkinter import filedialog
import matplotlib.pyplot as plt
import numpy as np
from scipy.fftpack import fft2, ifft2, fftshift

import tkinter as tk
from tkinter import filedialog
# Start up Tkinter
root = tk.Tk()
root.withdraw()

binary = target()
reference = target()
deconv = deconvolved()

# Define filenames for each target
binary.fits.fileName = filedialog.askopenfilename(title="Select Binary FITS file")
reference.fits.fileName = filedialog.askopenfilename(title="Select Reference FITS file")
binary.fits.fileName = "/home/niels/Documents/FITS/KP336.fits"
reference.fits.fileName = "/home/niels/Documents/FITS/KP338.fits"

# Import each target
binary.fits.read(numDimensions=3, printInfo=False)
reference.fits.read(numDimensions=3, printInfo=False)

# Calculate PSD of each target
```

83
binary.psdCalc()
print("Binary PSD Calc Complete")
reference.psdCalc()
print("Reference PSD Calc Complete")

## Deconvolve reference and binary stars with different methods

```
nxy = 512
center = int(nxy/2)
# Assign wiener variables
H = reference.psd.data
G = binary.psd.data
# Inverse filtering deconvolution
F_hat_inverse = G/H
f_hat_inverse = fftshift(np.abs(ifft2(F_hat_inverse)))
```

```
## Simplified Wiener filtering
# Set Constant
k1 = 1E-3
# Wiener filter and calculate acorr
F_hat_wiener1 = G*(1/H)*((H**2)/(H**2+k1))
f_hat_wiener1 = fftshift(np.abs(ifft2(F_hat_wiener1)))
```

```
## Wiener filtering with LPF for Signal PSD
# Setting constant and LPF radius
radius = 35
k2 = 3E-3
# Create centered meshgrid of image
xx,yy = np.meshgrid(np.arange(nxy),np.arange(nxy))
xx = np.subtract(xx,center)
yy = np.subtract(yy,center)
rr = np.power(np.power(xx,2)+np.power(yy,2),0.5)
# Create LPF filter image
lpf = np.exp(-(np.power(rr,2)/(2*np.power(radius,2))))
# Wiener filter and calculate acorr
F_hat_wiener2 = G*(1/H)*((H**2)/(H**2+k2/lpf))
f_hat_wiener2 = fftshift(np.abs(ifft2(F_hat_wiener2)))
```

```
## Deconvolving with simple LPF'ed inverse filter
deconv.psdDeconvolveLPF(psdBinary=G, psdReference=H, lpfRadius =20)
deconv.acorrCalc()
```

```
## Display Images
colormap = "jet"

plt.figure(figsize = (16,16), dpi = 100)
```
F. Python Classes for Speckle Data Processing

classes_labeyrie.py:

# Classes used in labeyrie speckle processing

# Module Includes
import matplotlib.pyplot as plt
import numpy as np
from scipy.fftpack import fft2, ifft2, fftshift
from astropy.io import fits
import sys, os
import ctypes

# Class to hold an array of data
# Built in methods for importing/exporting/viewing the data
# Used in target and deconvolved class
class fitsData():
    # Init
    def __init__(self):
        self.data = None # Holds data
        self.fileName = None # Holds filename for import/export
    # Read data from FITS file
    # Enter the number of dimensions of the FITS file to check for it on opening
    def read(self, numDimensions=2, printInfo=True, printHeaders=False): # Imports FITS file
        # Check if input file is .fits
        if (os.path.splitext(self.fileName)[1] != "".fits"):
            # Exit program if not FITS
            sys.exit("ERROR: " + self.fileName + " is not .fits")
        # Open FITS Data
        HDUList = fits.open(self.fileName)
        # Print FITS File Info & Headers
        if (printInfo == True):
            print(HDUList.info())
        if (printHeaders == True):
            print("Headers:")
            print(repr(HDUList[0].header))
        # Check that input psd FITS is appropriate dimension
        if (len(np.shape(HDUList[0].data)) != numDimensions):
sys.exit("ERROR: "+self.fileName+" dimensions 
  \n  != " + str(numDimensions))

# Save data in FITS cube to class's data variable, 
  \n  then close FITS file
self.data = HDUList[0].data
HDUList.close()

# Write data to FITS file
def write(self): # Write FITS file
  # Create PrimaryHDU object with data
  hdu = fits.PrimaryHDU(self.data)
  # Create HDUList object w/ PrimaryHDU
  hdulist = fits.HDUList([hdu])
  # Write to new file
  hdulist.writeto(self.fileName)

# View data
# Option to view log of data
# Option to select which image in 3D cube to view
def view(self, log=False, title=None, imNum=0):
  plt.figure()
  # Check for dimensionality of data
  # If 2D, show the image
  if len(np.shape(self.data)) == 2:
    if (log == False):
      plt.imshow(self.data)
    if (log == True):
      plt.imshow(np.log10(self.data))
  # If 3D, show the image of selected index
  elif len(np.shape(self.data)) == 3:
    if (log == False):
      plt.imshow(self.data[imNum])
    if (log == True):
      plt.imshow(np.log10(self.data[imNum]))
  # If other, must be an error
  else:
    sys.exit("Can only view 2D or 3D data")
  plt.title(title)
  plt.colorbar()
  plt.show()

# Target Class: Holds data for a Reference or Binary Star
class target():
  # Init
def __init__(self):
self.fits = fitsData()  # Holds raw astronomical data
self.psd = fitsData()  # Holds calculated PSD of target

def psdCalc(self):
    # Calculate PSD of FITS data
    # Checking if FITS data is an array of images
    if (len(self.fits.data.shape) == 3):
        # Generate empty array the size of an image to be used to accumulate
        # PSD values before averaging.
        psdShape = (self.fits.data.shape[1], int(self.fits.data.shape[1]/2+1))
        psdSum = np.zeros(psdShape, dtype=np.float32)
        psdAvg = np.zeros(psdShape, dtype=np.float32)

        imgNum = np.shape(self.fits.data)[0]  # Number of images
        imgIncrement = imgNum/20  # How often to display a status message

        # Looping through all images in cube
        for index, img in enumerate(self.fits.data):

            # Print current file being processed
            if (((index+1) % imgIncrement) == 0):
                print("Processed Image #: ",(index+1), "/", imgNum)

            # Calculate 2D power spectrum
            # This gives us only real values
            psdImg = fftw_psd(img)

            # Accumulate current PSD value
            psdSum = np.add(psdSum, psdImg)

        # Divide by # of images to calculate average
        psdAvg = np.divide(psdSum, imgNum)

        # Normalizing FFT
        psdAvg = np.divide(psdAvg, (psdAvg.size)**2)
    
    # Otherwise if FITS data is only one image
    elif (len(self.fits.shape) == 2):
# Calculate 2D power spectrum
# This gives us only real values
psdImg = fftw_psd(img)

# Normalizing FFT
psdAvg = np.divide(psdImg, (psdImg.size)**2)

self.psd.data = fftshift(transpose_fftw_psd(psdAvg)).astype(np.float32)

# Deconvolved Class: Holds data for devonvolved targets
class deconvolved():
    # Init
    def __init__(self):
        self.psd = fitsData()  # Holds deconvolved PSD
        self.acorr = fitsData()  # Holds autocorrelation

    # Deconvolve PSDs
    def psdDeconvolveWiener(self, psdBinary, psdReference, k=None, lpfRadius=None):
        imgSize = np.shape(psdReference)[0]  # Calculate dimension of image
        imgCenter = int(imgSize/2)  # Center index of image

        # Create centered meshgrid of image
        xx, yy = np.meshgrid(np.arange(imgSize), np.arange(imgSize))
        xx = np.subtract(xx, imgCenter)
        yy = np.subtract(yy, imgCenter)
        rr = np.power(np.power(xx, 2) + np.power(yy, 2), 0.5)

        # Create LPF filter image if specified
        if (lpfRadius != None):
            lpf = np.exp(-(np.power(rr, 2)/(2*np.power(lpfRadius, 2))))
        else:
            lpf = np.zeros((imgSize, imgSize))
            lpf.fill(1)

        # Perform wiener filtering
        self.psd.data = psdBinary*(1/psdReference)*((psdReference**2)/(psdReference**2+k/lpf))
def psdDeconvolveLPF(self, psdBinary, psdReference, lpfRadius=None):
    imgSize = np.shape(psdReference)[0]  # Calculate dimension of image
    imgCenter = int(imgSize/2)  # Center index of image
    # Create centered meshgrid of image
    xx, yy = np.meshgrid(np.arange(imgSize), np.arange(imgSize))
    xx = np.subtract(xx, imgCenter)
    yy = np.subtract(yy, imgCenter)
    rr = np.power(np.power(xx,2) + np.power(yy,2), 0.5)
    # Create LPF filter image if specified
    lpf = np.exp(-(np.power(rr,2)/(2*np.power(lpfRadius,2))))
    # Perform wiener filtering
    self.psd.data = psdBinary * (1/psdReference) * lpf

    # Calculate autocorrelation from PSD
    def acorrCalc(self):
        # Because we normalized after FFT by multiplying by 1/N^2, and ifft
        # function does this as well, we need to multiply by N^2 before ifft
        # to prevent performing normalization twice
        self.acorr.data = self.psd.data *(self.psd.data.size)
        # Do iFFT on PSD’s, bringing back to spatial domain
        # This should give us the autocorrelations of original images
        self.acorr.data = ifft2(self.acorr.data)

        # Taking iFFT of PSD (all real values) results in complex valued output
        # Must view the magnitude of the output
        # Doing FFTshift to move eyes of autocorrelation near center
        # Taking iFFT of PSD (all real values) results in complex valued output
        # Must view the magnitude of the output
        self.acorr.data = np.abs(fftshift(self.acorr.data))

        # Use FFTW to calculate the PSD of a single image

90
# Input image = 512x512 ndarray of np.double32
# Output image = 512x257 ndarray of np.double32

def fftw_psd(input_img):
    # Import shared C library
    fftw_psd_dll = ctypes.CDLL('/home/niels/Dropbox/Thesis/
                             Python/fftw_psd.so')
    # Calculating imgsize parameters
    # Size of image Height
    imgsize = 512
    # Number of pixels in PSD
    psd_n = imgsize*(int(imgsize/2)+1)
    # Number of pixels in IMG
    img_n = imgsize**2

    # Reshape Square array to be flat
    input_img_flat = np.reshape(input_img.astype(np.float32), (imgsize**2, 1))

    # Create pointers for in/out
    img_ptr = (input_img_flat).ctypes.data_as(ctypes.POINTER(
                             ctypes.c_float))
    out_ptr = (np.zeros(img_n, np.float32)).ctypes.data_as(
                             ctypes.POINTER(ctypes.c_float))

    # Array type to be passed to wrapped C function
    # Set input argument to be flat array of doubles (# of
    # input img pixels)
    fftw_psd_dll.psd.argtypes = [ctypes.POINTER(ctypes.c_float)
                             ]
    fftw_psd_dll.psd.restype = ctypes.POINTER(ctypes.c_float * 
                             psd_n)

    # Calculate PSD, get a pointer returned
    out_ptr = fftw_psd_dll.psd(img_ptr)

    # Reshape array to image
    psd_image = np.reshape(out_ptr.contents,(imgsize,int(
                             imgsize/2+1)))

    # Return PSD Image
    return psd_image

# Using FFTW to calculate PSD generates a 512x257 nonredundant
# array
# Need to transpose this array into a redundant 512x512 array
# to take iFFT

def transpose_fftw_psd(fftw_psd_img):
    # Calculate Image Dimensions
imgsiz = np.shape(fftw_psd_img)[0]

# Create a padded version of original image
fftw_psd_img_padded = np.lib.pad(fftw_psd_img, ((0, 0), (0, int(imgsize/2-1))), 'constant')

# Loop through first row
y = 0
y_source = 0
for x in np.arange(int(imgsize/2)+1, imgsize):
    x_source = imgsize-x
    fftw_psd_img_padded[y,x] = fftw_psd_img_padded[y_source,x_source]

# Loop through rest of image
for y in np.arange(1, imgsize):
    for x in np.arange(int(imgsize/2)+1, imgsize):
        y_source = imgsize-y
        x_source = imgsize-x
        fftw_psd_img_padded[y,x] = fftw_psd_img_padded[y_source,x_source]

return fftw_psd_img_padded

classes.astrometry.py:

# Module Includes
import matplotlib.pyplot as plt
import numpy as np
from scipy.fftpack import fft2, ifft2, fftshift
from astropy.io import fits
import sys, os, cv2
import ctypes

# coords class: Holds methods for working in polar/cart
#ordinates
# Used in camsky class
class coords():
    def __init__(self, midpoint):
        self.theta = None
        self.rho = None
        self.x = None
        self.y = None
        self.midpoint = midpoint

    # Orbital plot has 0deg pointing down on the screen [
    # resulting in 90deg shift]
    # Theta rotates counterclockwise from 0deg
# Positive pixel direction is downward [resulting in Y
 phase reversal]
# Polar coordinates centered on middle pixel of image

# convert theta and rho values to image x and y
coordinates

def polar2cart(self):
    self.x = self.midpoint + (self.rho * np.cos(np.radians
             (self.theta-90)))
    self.y = self.midpoint - (self.rho * np.sin(np.radians
             (self.theta-90)))

# convert x and y coordinates to theta and rho values
cart2polar(self):
    # Get cartesians centered on 0 for calculations
    x = self.x-self.midpoint
    y = self.midpoint-self.y

    self.rho = np.linalg.norm((x,y))
    self.theta = np.degrees(np.arctan2(y,x))+90
    # Add 360 deg if theta is a negative angle
    self.theta = self.theta + 360*(self.theta<0)

# camsky class: Holds methods for working between measurements
in camera and sky
class camsky():
    def __init__(self, midpoint, delta, e):
        self.cam = coords(midpoint)
        self.sky = coords(midpoint)
        self.delta = delta
        self.e = e

    # Convert from polar coords in sky to polar coords in
    camera
def sky2cam(self):
        self.cam.theta = self.sky.theta+self.delta
        self.cam.rho = self.sky.rho/self.e

    # Convert from polar coords in camera to polar coords in
    sky
def cam2sky(self, camTheta, camRho):
        self.sky.theta = self.cam.theta-self.delta
        self.sky.rho = self.cam.rho*self.e

# Astrometry class: Holds data for astrometry measurements
class astrometry():
    def __init__(self,):
```python
def __init__(self):
    self.acorr = None  # Raw Autocorrelation
    self.acorrMarked = None  # Marked up Autocorrelation

# Clear marked up acorr back to original acorr
def acorrMarkedClear(self):

    # Create blank RGB image
    imgDim = np.shape(self.acorr)[0]
    self.acorrMarked = np.zeros((imgDim, imgDim, 3), np.uint8)

    # Calculate scaled version of autocorr image
    imgScaled = np.divide(self.acorr, np.max(self.acorr))
    imgScaled = (np.multiply(imgScaled, 255)).astype(np.uint8)

    # Fill in scaled image into each R/G/B Channel
    for i in np.arange(3):
        self.acorrMarked[:, :, i] = imgScaled

    # Mark acorr at location with indicated shape
    def acorrMark(self, x, y, shape, color, radius=10):
        if shape == 'o':
            cv2.circle(self.acorrMarked, (x, y), radius, color, 1)
        elif shape == '+':
            cv2.line(self.acorrMarked, (x- radius, y), (x+ radius, y), color, 1)
            cv2.line(self.acorrMarked, (x, y-radius), (x, y+radius), color, 1)
        else:
            print("Invalid shape input")

    # centroidExpected[x,y]: Expected location of secondary area
    # centroidCalculate(x,y,radius): Calculate centroid within area

    # Estimate centroids of autocorr
    # Autocorrelation image consists primarily of the large central lobe and less tall two side lobes
    # Lobes are found by creating a thresholded image, 1 where above threshold otherwise 0
    # Threshold is started at peak of center lobe, moved down to the minimum value that gives 3 contours
    def centroidEstimate(self):
```

94
# Start thresh at max value of autocorr (assuming that
# is from central peak)
threshold = np.floor(self.acorr.max())

# Calculate values to increment/decrement threshold by
# depending on height of autocorrelogram
incrementFine = self.acorr.max()/100

# Assume we start with 1 contour visible
numContours = 1

# Step down threshold until we see the 3 contours we
# expect
while (numContours != 3):
    # Decrement the threshold
    threshold = threshold - incrementFine

    # Create thresholded image
    acorrThresh = self.acorr.data > threshold #
    acorrThresh = (np.multiply(acorrThresh,255).astype
    (np.uint8))

    # Find contours of threshold image
    im, contours, hierarchy = cv2.findContours(
        acorrThresh,cv2.RETR_TREE,cv2.
        CHAIN_APPROX_SIMPLE)
    numContours = len(contours)

    if threshold < 0:
        print("Error in finding 3 contours")
        sys.exit()

# Now we’ve found our 3 main contours. Keep stepping
# down until we don’t see 3 anymore
# Can be less than 3 if they start to blend together,
# or more than 3 if other noise in
# image appears
while (numContours == 3):
    # Decrement the threshold
    threshold = threshold - incrementFine

    # Create thresholded image
    acorrThresh = self.acorr.data > threshold #
    acorrThresh = (np.multiply(acorrThresh,255).astype
    (np.uint8))
# Find contours of threshold image
im, contours, hierarchy = cv2.findContours(acorrThresh, cv2.RETR_TREE, cv2.CHAIN_APPROX_SIMPLE)
numContours = len(contours)

if threshold < 0:
    print("Error in moving threshold below 3 contours")
    sys.exit()

# We've moved just below our desired 3 contours. We moved down by incrementFine, so we need to move up by incrementFine to get back to the proper threshold
threshold = threshold + incrementFine
# Create thresholded image
acorrThresh = self.acorr.data > threshold # Calculate indices in image above threshold
acorrThresh = (np.multiply(acorrThresh, 255).astype(np.uint8))
# Find contours of threshold image
im, contours, hierarchy = cv2.findContours(acorrThresh, cv2.RETR_TREE, cv2.CHAIN_APPROX_SIMPLE)
numContours = len(contours)

# Now that we have the contours, we want to find their centroids for sorting purposes
centroid = np.zeros((3, 2))
centerDistance = np.zeros(3)
imgCenter = np.divide(np.shape(self.acorr), 2) # Center pixel indices

for i in np.arange(3):
    # Calculating Image Moment/Raw Moment of contour
    M = cv2.moments(contours[i])

    # Calculate centroid X and Y components
cx = int(M['m10']/M['m00'])
cy = int(M['m01']/M['m00'])

    # Save centroid
    centroid[i] = (cx, cy)

    # Calculate distance from centroid to image center
centerDistance[i] = np.linalg.norm(centroid[i] - imgCenter)

# Get index of central contour as contour with min distance to center
iCenter = np.where(centerDistance == centerDistance.min())
iCenter = iCenter[0]

# Finding indices of side lobes
iSide = [0,1,2] # Possible indices of side lobes
iSide.remove(iCenter) # Remove the center lobe index

# Make empty mask images
maskSide = np.zeros((2, np.shape(self.acorr)[0], np.shape(self.acorr)[1]))
lobeSide = np.zeros((2, np.shape(self.acorr)[0], np.shape(self.acorr)[1]))

# Create masks and calculate side lobes
for i in np.arange(2):
    cv2.drawContours(maskSide[i], contours, iSide[i], (1,1,1), -1)
    lobeSide[i] = (np.multiply(self.acorr, maskSide[i]))

# Calculating intensity weighted centroid of side lobes, using entire contour
M00 = np.zeros((2))
M10 = np.zeros((2))
M01 = np.zeros((2))
centroid = np.zeros((2,2))

# Calculate for each side lobe
for lobe in np.arange(2):

    # Calculate average
    M00[lobe] = np.sum(lobeSide[lobe])

    # Calculate X component
    # Loop through each column
    for column in np.arange(np.shape(lobeSide[lobe])[1]):
        M10[lobe] += np.multiply(column, np.sum(lobeSide[lobe,:], column))

    # Calculate Y component
# Loop through each row
for row in np.arange(np.shape(lobeSide[lobe])[0]):
    M01[lobe] += np.multiply(row, np.sum(lobeSide[
        lobe, row, :]))

# Calculate X and Y Centroid Components of each lobe
centroid[lobe, 0] = M10[lobe] / M00[lobe]  # X Component

# Return centroid locations
return centroid
G. Processing FITS Data using classes_labeyrie.py

Preprocessing done in labeyrie.preprocess.py:

```
# Calculates PSD of input FITS files

# Import Modules
import sys, os
from classes_labeyrie import target
import tkinter as tk
from tkinter import filedialog
import numpy as np
import matplotlib.pyplot as plt

# Raw data target object
raw = target()

# Start up Tkinter
root = tk.Tk()
root.withdraw()

# Prompt for FITS file locations
fitsFileNames = filedialog.askopenfilenames(title="Select FITS files for processing")

# Debug
# fitsFileNames = []
# fitsFileNames.append('/home/niels/Desktop/KP330.fits')
# fitsFileNames.append('/home/niels/Desktop/KP331.fits')
# fitsFileNames.append('/home/niels/Desktop/KP332.fits')

# Loop through each fileName
for fitsFileName in fitsFileNames:
    # Import FITS file
    raw.fits.fileName = fitsFileName
    raw.fits.read(numDimensions=3, printInfo=False)

    # Create new filename
    psdFileName = os.path.splitext(fitsFileName)[0]
    psdFileName = psdFileName + "_PSD.fits"

    # If filename already exists, add "0" to the end of filename
    # until no longer is a repeat
```
# Safety condition for this loop : break out if more than 4 repeats
fileRepeat = 0
while ( os.path.exists(psdFileName) ):
    # Check if too many repeated filenames
    if (fileRepeat > 3):
        print("Too many repeated filenames, delete some")
        break
    # Add a 0 to filename
    psdFileName = os.path.splitext(psdFileName)[0]
    psdFileName = psdFileName + "0. fits"
    fileRepeat += 1

if (os.path.exists(psdFileName)):
    # if no good filename found
    # Don’t generate file
    print("No processed file generated")

else:
    print("Processing file: ", fitsFileName)
    print("Creating file: ", psdFileName)

    # Process FITS data
    raw.psdCalc()

    # Create new FITS file
    raw.psd.fileName = psdFileName
    raw.psd.write()

    #Print message for user
    print("Done processing ", psdFileName)
    print()
    print()

Deconvolution done in labeyrie_deconv.py:

# Module for performing Labeyrie deconvolution of preprocessed FITS files

# Included modules
import sys
from classes_labeyrie import target, deconvolved
from classes_astrometry import astrometry, camsky
import tkinter as tk
import matplotlib.pyplot as plt
import numpy as np
import math
from tkinter import filedialog
import os, sys

# Start up Tkinter
root = tk.Tk()
root.withdraw()

# Instantiate objects
binary = target()
reference = target()
deconv = deconvolved()
calcs = astrometry()

# Prompt user for binary PSD file location
binary.psd.fileName = filedialog.askopenfilename(title="Select BINARY FITS file")
#binary.psd.fileName = "/home/niels/Documents/FITS/KP336_PSD.fits"
# Import binary star PSD FITS
binary.psd.read()

# Prompt user for reference PSD file location
reference.psd.fileName = filedialog.askopenfilename(title="Select REFERENCE FITS file")
#reference.psd.fileName = "/home/niels/Documents/FITS/KP338_PSD.fits"
# Import reference star PSD FITS
reference.psd.read()

# Deconvolve reference and binary stars
#deconv.psdDeconvolveWiener(binary.psd.data, reference.psd.data, lpfRadius = 30, k=1e-7)
deconv.psdDeconvolveLPF(binary.psd.data, reference.psd.data, lpfRadius = 20)

# Get autocorrelogram
deconv.acorrCalc()

# Normalize acorr data to between 0 and 1
deconv.acorr.data = np.divide(deconv.acorr.data, deconv.acorr.data.max())
# View Results
deconv.acorr.view(title="Autocorrelation")
# Write acorr to file
deconv.acorr.fileName = ''
deconv.acorr.fileName = filedialog.asksaveasfilename(
    defaultextension='.fits', initialdir=(os.path.split(
        reference.psd.fileName)[0]+'/'))
if (deconv.acorr.fileName != ''):
    deconv.acorr.write()

# Displaying estimated observed and expected locations
if (input("Perform centroid calculation? [y/n] ").lower() == 
    'n'):
    sys.exit("Done")
delta = float(input("Enter camera angle (delta) in degrees "))
e = float(input("Enter plate scale (e) in arcsec/pixel "))
# Calculate middle index of image
midpoint = np.shape(deconv.acorr.data)[0]/2
# Create object for observed and expected secondary locations
obs0 = camsky(midpoint=midpoint, delta=delta, e=e)
obs1 = camsky(midpoint=midpoint, delta=delta, e=e)
exp = camsky(midpoint=midpoint, delta=delta, e=e)

# Input expected secondary location to be displayed
exp.sky.theta = float(input("Enter secondary’s expected 
    location angle (theta) in degrees "))
exp.sky.rho = float(input("Enter secondary’s expected location 
    separation (rho) in arcsec "))
exp.sky2cam()
exp.cam.polar2cart()

# Testing centroid estimation
calcs.acorr = deconv.acorr.data # Moving acorr to astrometry
centroid = calcs.centroidEstimate()
# Save centroid locations to observed position objects
(obs0.cam.x,obs0.cam.y)=centroid[0]
(obs1.cam.x,obs1.cam.y)=centroid[1]

# Viewing estimated centroid locations
# Prepare the object for marking up with locations
calcs.acorr = deconv.acorr.data
calcs.acorrMarkedClear()
# Mark expected position
calcs.acorrMark(exp.cam.x.astype(np.uint16),exp.cam.y.astype(
    np.uint16),"o",(0,255,0))
# Mark observed position
calcs.acorrMark(obs0.cam.x.astype(np.uint16), obs0.cam.y.astype(np.uint16), "o", (255, 0, 0))
calcs.acorrMark(obs1.cam.x.astype(np.uint16), obs1.cam.y.astype(np.uint16), "o", (255, 0, 0))

# View Image
plt.figure()
plt.imshow(calcs.acorrMarked)
plt.title("Observed and expected secondary locations")
plt.show()
H. Benchmarking PSD Computation Methods

Benchmark of PSD methods in *bench.psd_calc.py*

# Calculates PSD with several methods and compares calculation times

# Import Modules
import sys, os
from classes_labeyrie import target, fftw_psd
import tkinter as tk
from tkinter import filedialog
from cython_psd import cython_psd

import numpy as np
import scipy as sp
import matplotlib.pyplot as plt
import time

# Raw data target object
raw = target()

# Start up Tkinter
root = tk.Tk()
root.withdraw()

# Prompt for FITS file locations
raw.fits.fileName = filedialog.askopenfilename(title="Select FITS file for test data")
# raw.fits.fileName = "/home/niels/Documents/FITS/KP330.fits"

# Import FITS data
raw.fits.read(numDimensions=3)
image = raw.fits.data[0]

# Number of iterations to do for benchmark
iterations = 100

# Labels for data
time_data = {}

# Test Numpy PSD Calculation
avg_time = 0
for i in np.arange(iterations): # Repeat
    ts = time.time() # Start Timer
    # Perform calculation
    ts = time.time() # Stop Timer
    avg_time += ts - ts

# Average time
avg_time /= iterations

# Additional code for other methods can be added here.
psd = np.power(np.abs(np.fft.fft2(image)),2) # Execute Function
te = time.time() # Stop timer
avg_time += (te-ts)/iterations # Accumulate time
time_data["NumPy FFT"] = avg_time*1000

# Test SciPy PSD Calculation
avg_time = 0
for i in np.arange(iterations): # Repeat
    ts = time.time() # Start Timer
    psd = np.power(np.abs(sp.fftpack.fft2(image.astype(np.float32))),2) # Execute Function
    te = time.time() # Stop timer
    avg_time += (te-ts)/iterations # Accumulate time
time_data["SciPy FFTpack"] = avg_time*1000

# Test Optimized Scipy PSD Calculation
avg_time = 0
for i in np.arange(iterations): # Repeat
    ts = time.time() # Start Timer
    psd = np.abs(sp.fftpack.fft2(image.astype(np.float32)))**2 # Execute Function
    te = time.time() # Stop timer
    avg_time += (te-ts)/iterations # Accumulate time
time_data["Modified SciPy FFTpack"] = avg_time*1000

# Test Cython NumPy PSD Calculation
avg_time = 0
for i in np.arange(iterations): # Repeat
    ts = time.time() # Start Timer
    psd = cython_psd(image) # Execute function
    te = time.time() # Stop timer
    avg_time += (te-ts)/iterations # Accumulate time
time_data["Cython NumPy FFT"] = avg_time*1000

# Test FFTW PSD Calculation
avg_time = 0
for i in np.arange(iterations): # Repeat
    ts = time.time() # Start Timer
    psd = fftw_psd(image) # Execute Function
    te = time.time() # Stop timer
    avg_time += (te-ts)/iterations # Accumulate time
time_data["Ctypes FFTW"] = avg_time*1000

# Print Data
for name, value in time_data.items():
    print('{0:25} ==> {1:10.2f}ms'.format(name, value))
# Graph Data
ind = np.arange(len(time_data))
width = 0.35
fig = plt.figure()
ax = fig.add_subplot(111)
ax.bar(ind, time_data.values())
ax.set_xticklabels(time_data.keys(), rotation=45)
ax.set_xticks(ind + width/2)
ax.set_ylabel("Execution Time [ms]")
ax.set_title("Execution Time Comparison of Different PSD Calculations")
ax.grid(True)
plt.tight_layout()
plt.show()

Cython PSD Function generated from cython_psd.pyx file

# Cython module for calculating PSD
cimport numpy as np
import numpy as np

def cython_psd(np.ndarray image):
    cdef np.ndarray psd = np.zeros([512,512], dtype=np.float32)

    # Calculate 2D power spectrum
    # This gives us only real values
    psd = np.abs(np.fft.fft2(image.astype(np.float32)))**2

    return psd

Optimized FFTW C Code in fftw_psd.c

#include <stdlib.h>
#include <stdio.h>
#include <fftw3.h>
#include <time.h>

//#define PRINT_DATA

#define IMG_SIZE 512

// Compile with :
// Using makefile : make -f fftw_psd_makefile
// or by hand
// gcc -o fftw_psd.so -shared -fPIC fftw_psd.c
float * psd(float * in_img);
float * psd2(float * in_img);

// Main tests the timing of PSD function
void main(void) {
    int i;
    float *in;
    float *out_psd;
    int j;
    int nx = IMG_SIZE;
    int ny = IMG_SIZE;
    int nyh = ( ny / 2 ) + 1;
    unsigned int seed = 123456789;
    unsigned int max_input = 1024;

    /*
     Create the input array, an NX by NY array of floats.
    */
    in = ( float * ) malloc ( sizeof ( float ) * nx * ny );
    out_psd = ( float * ) malloc ( sizeof ( float ) * nx * nyh );

    /*
    srand ( seed );
    */
    for ( i = 0; i < nx; i++ )
    {
        for ( j = 0; j < ny; j++ )
        {
            in[i*ny+j] = rand ()%max_input;
        }
    }

    /*
    in[4*ny+5] = 1;
    in[4*ny+3] = 1;
    */
    /*
    Save start time
    */
    clock_t tic = clock();

    /*
    Execute
    */
    out_psd = psd(in);
/* 
Save FFT time 
*/

clock_t toc = clock();

/*/ 
Print Data 
*/

printf ("\n");
printf ("TEST04\n");
printf (" Demonstrate FFTW3 on a %d by %d array of real 
 → data.\n", nx, ny);
printf ("\n");
printf (" Transform data to FFT coefficients.\n" );
printf (" Backtransform FFT coefficients to recover data.\n → n" );
printf (" Compare recovered data to original data.\n" );

printf("\n\n");
printf(" Total Computation Time : \n");
printf(" Elapsed: %f seconds\n", (float)(toc - tic) / 
 ←CLOCKS_PER_SEC);

#ifdef PRINT_DATA
 printf ("\n");
 printf (" Input Data:\n");
 printf ("\n");

 for ( i = 0; i < nx; i++ )
 { 
    for ( j = 0; j < ny; j++ )
    {
        printf ( " %4d %4d %12f\n", i, j, in[i*ny+j] );
    }
 }

 printf ("\n");
 printf (" Output PSD:\n");
 printf ("\n");

 for ( i = 0; i < nx; i++ )
 { 
    for ( j = 0; j < nyh; j++ )
    {
        printf ( " %4d %4d %12f \n", 

108
float * psd(float * in_img) {
    // Counters
    int i;
    int j;

    // Time holders
    clock_t tic, toc;

    // Img Dimensions
    int nx = IMG_SIZE;
    int ny = IMG_SIZE;
    // PSD has symmetry about the center,
    // only need to calculate half the image
    int nyh = (ny / 2) + 1;
    double psd;

    // Allocate memory for fft output/plans
    fftw_complex *out;
    double in_img_double[IMG_SIZE*IMG_SIZE];
    static float out_psd[IMG_SIZE*(IMG_SIZE/2+1)];
    fftw_plan plan_forward;

    for (i = 0; i < nx; i++) {
        for (j = 0; j < ny; j++) {
            in_img_double[i*ny+j] = (double)in_img[i*ny+j];
        }
    }

    /*
    Create the output array OUT, which is of type FFTW_COMPLEX,
    and of a size NX * NYH
```c
/*
 out = fftw_malloc ( sizeof ( fftw_complex ) * nx * nyh );
/*
 Set up and execute FFTW Plan
 */
plan_forward = fftw_plan_dft_r2c_2d ( nx , ny , in_img_double ,
 out , FFTW_ESTIMATE );
fftw_execute ( plan_forward );
/*
 Calculate PSD of complex output
 */
for ( i = 0; i < nx; i++ )
{
 for ( j = 0; j < nyh; j++ )
{
  // out_psd[i*nyh+j] = 3.0;
  psd = ((out[i*nyh+j][0]*out[i*nyh+j][0]) + (out[i*nyh+j
   [1]*out[i*nyh+j][1]));
  // printf("Current index = %d, ", i*nyh+j);
  // printf("Present PSD (double) = %f, ", psd);
  // printf("Present PSD (float) = %f, ", (float)psd);
  // printf("\n");
  out_psd[i*nyh+j] = (float)psd;
}
}
/*
 Free up the allocated memory and return PSDs
 */
fftw_destroy_plan ( plan_forward );
fftw_free ( out );
return out_psd;
}
float * psd2(float * in_img) {
  // Counters
  int i;
  int j;

  // Time holders
  clock_t tic,toc;

  // Img Dimensions
```
int nx = IMG_SIZE;
int ny = IMG_SIZE;
double psd;

// Allocate memory for fft output/plans
fftw_complex *in;
fftw_complex *out;
static float out_psd[IMG_SIZE*(IMG_SIZE/2+1)];

in = fftw_malloc ( sizeof ( fftw_complex ) * nx * ny );
out = fftw_malloc ( sizeof ( fftw_complex ) * nx * ny );

// Make FFTW Plan
fftw_plan plan_forward;
plan_forward = fftw_plan_dft_2d ( nx , ny , in , out,
                                 FFTW_FORWARD, FFTW_ESTIMATE );

// Copy input data to complex array
for ( i = 0; i < nx; i++ )
{
    for ( j = 0; j < ny; j++ )
    {
        in[i*ny+j][0] = (double)in_img[i*ny+j];
    }
}

/*
 Execute FFTW Plan
*/
fftw_execute ( plan_forward );

/*
 Calculate PSD of complex output
*/
for ( i = 0; i < nx; i++ )
{
    for ( j = 0; j < ny; j++ )
    {
        // out_psd[i*ny+j] = 3.0;
        psd = ((out[i*ny+j][0]*out[i*ny+j][0]) + (out[i*ny+j][0]*out[i*ny+j][1]));
        // printf("Current index = %d, ", i*ny+j);
        // printf("Present PSD (double) = %f, ", psd);
        // printf("Present PSD (float) = %f, ", (float)psd);
        // printf("\n");
        out_psd[i*ny+j] = (float)psd;
    }
}
} } 

/*
 Free up the allocated memory and return PSDs
 */

fftw_destroy_plan ( plan_forward );
fftw_free ( in );
fftw_free ( out );

return out_psd;
}