

Use of a Digital Micromirror Array as a Configurable Mask in Optical Astronomy

Shawn Gilliam

Department of Physics
301 Weniger Hall
Oregon State University
Corvallis, OR 97331-6507
USA

Advisor: Dr. William Hetherington

Abstract

High resolution images of a dim companion in a binary star system can be obtained using a Digital Micromirror Array (DMA) as an optical mask. Imaging can be problematic because the intense light from the main star can saturate the detector and leave the companion unnoticed. Placement of the DMA in the image plane allows for the replication of widely-used optical masking techniques. A laboratory model of a binary star system with a bright source that is about 5000 times brighter than the dim companion is resolved at 25 ± 5 arcseconds. Another simulation uses the pixels of a monitor to acquire an angular resolution of 3.4 ± 0.1 arc seconds. The system uses a 12" Cassegrain reflecting telescope to focus light onto the 1024 x 768 pixel D4100 DMA, the light is then taken from the masked image plane to a charge coupled device (CCD) via a 150 mm lens. Lyot Stop masks and Aperture Masking Interferometry will also be discussed.

Table of Contents

Introduction	3
Instrumentation	3
System Calibration	5
Prior Applications.....	5
Coronagraphs	7
Lyot Stop Mask.....	8
Aperture Masking Interferometry.....	8
Experimental Results.....	11
E1. “Lab Stars” Experiment	11
E2. “Hall Star” Experiment	13
E3. “Monitor Stars” Experiment.....	15
Discussion:	16
Conclusion:.....	18
References.....	19
Appendix A - Software	20
Python Code.....	21
Appendix B – Fourier Transform Laboratory Exercise	41
Appendix B – Fresnel Zone Plates Laboratory Exercise.....	45

Introduction

This paper will discuss the abilities of a Digital Micromirror Array (DMA) in astronomy as an optical masking device. An exploration of some of the current applicable techniques and the theory behind each method are discussed. Possible implementation of the techniques using the DMA will be examined. Laboratory tests are then conducted with stellar object simulations to determine the abilities of the DMA to function as a useful tool in astronomy. The results of the experiments are considered and compared with expected outcomes.

Instrumentation

The DMA, or Digital Light Processing (DLP) Discovery D4100 .55" Extended Graphics Array (XGA) Digital Micromirror Device, is a highly manipulatable array of mirrors. In the array, there are 1024 columns and 768 rows. The pitch or distance from center to center is $10.8\text{ }\mu\text{m}$, with about $1\text{ }\mu\text{m}$ space between them. Each mirror can be turned "ON" or "OFF", or $+12^\circ$ and -12° respectively, from normal, as shown in Figure 1. Projection screens for some televisions, video projectors, and many other applications are based on this technology.

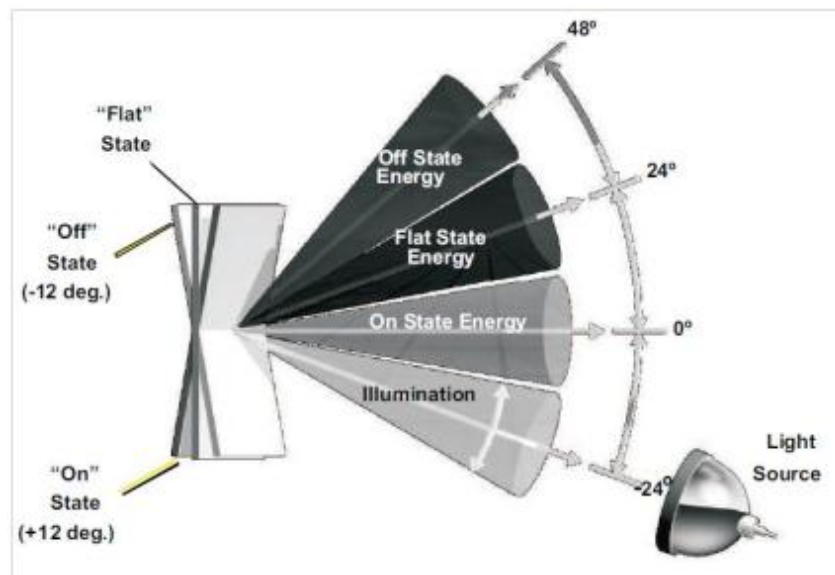


FIGURE 1: DIAGRAM OF POSSIBLE MIRROR POSITIONS. THE MIRRORS CAN TILT 12 DEG. IN EITHER DIRECTION, DIVERTING THE LIGHT AWAY OR TOWARD THE DETECTOR. [6]

The data required to encode the DMA with a pattern are in the monochrome bitmap format. Monochrome bitmaps are arrays of numbers that begin with a heading that describes the array. The heading provides information about the file such as colors, size, and when to begin reading the data that result in a pattern. More information on this is available in various places on the internet. [8] The technically important aspect of the bitmap is that the ones and zeros which correspond to white and black, respectively, determine which direction each mirror will face. In this configuration the white is off and

the black is on.

The DMA apparatus is comprised of four pieces of equipment. Starting from the perspective of the light path, the first instrument is a Meade LX200 12" Cassegrain telescope. The lens for this telescope has an $f/10$ focal ratio, and a resolving power of 0.380 arcseconds. [9] After the light passes through the telescope it will reflect off of the DMA. The DMA as a separate system is reported to have an 88% reflectivity (nominal 68% efficiency) in the 420 – 700 nm spectrums. [6] The DMA will be placed in the image plane of the telescope in order to manipulate the image and divert the unwanted light. A lens with a 150 mm focal length is placed into the optical path which then relays the image onto the CCD where the image is displayed on a computer monitor. A program called Discovery 4100 Explorer was supplied with the DMA; this program loads image patterns onto the array. A series of Python programs have been written to construct the images that are used for analysis. The code and a short description of the more useful programs are located in Appendix A for more information.

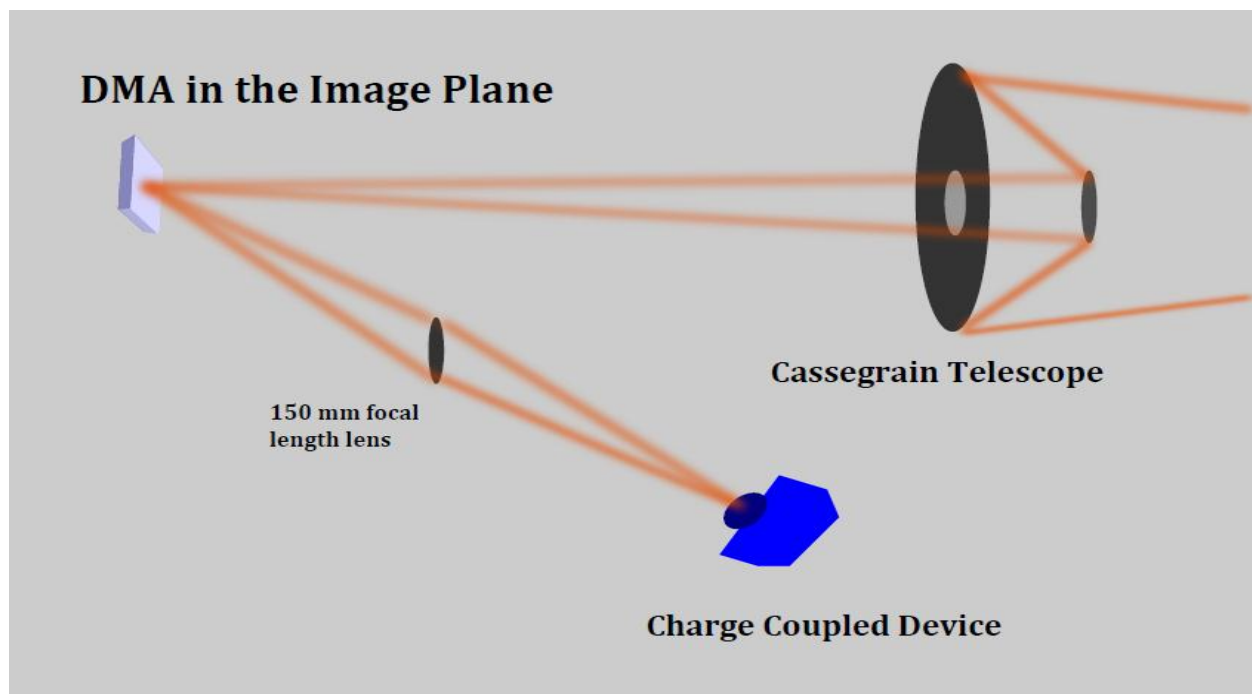


FIGURE 2: DMA APPARATUS DIAGRAM. THE LIGHT ENTERS THROUGH THE TELESCOPE AND IS THEN INCIDENT ON THE DMA AT THE IMAGE PLANE. THE LIGHT THEN TRAVELS THROUGH A LENS AND ON TO THE CCD.

System Calibration

Analysis of the optical assembly is provided by a computation of the Optical Transfer Function (OTF) of the apparatus. The optical transfer function describes the quality of the optical system. Aberrations in the system, which lower the quality of the image, will carry through the optical train and be multiplied by the subsequent components. The point spread function (psf) is an elementary flux-density pattern that describes the translation of a point source as it passes through an optical system. Information found in the OTF describes asymmetrical aberrations such as coma and astigmatism when present. This is found in the phase transfer function describing linear shift in the intensity pattern. The modulation transfer function describes the level of contrast that is delivered to the image from the object through the optical train. The point spread function, phase transfer function, and modulation transfer functions are all embedded in the optical transfer function.

Using the Convolution theorem in the spatial frequency domain, the Fourier transform of the image is equal to the product of the Fourier transform of the object irradiance and the Fourier transform of the point spread function. By dividing the image transform by the object transform the OTF is obtained.

$$\frac{F\{I_i(X,Y)\}}{F\{I_o(x,y)\}} = F\{psf(x,y)\} = OTF$$

The Fourier transform of the point spread function is the OTF, and is computed for this system using a python program created specifically for this experiment. [See Appendix A OTFfinder] The program prompts the user for an image, object image, and specifications for the Fourier transforms. The images are put into arrays of numbers. The transformed image is divided by the object transform and they form another array that is then plotted as a two dimensional Fourier transform that describes the OTF.

Prior Applications

The DMA is useful as an optical mask in the Fourier Transform plane. Using an appropriate lens, the Fourier Transform of an image can be placed precisely onto the plane of the mirror pixels. This allows the individual frequencies to be removed or transmitted as directed by the mask. An example of the results is shown in figure 2. The left image shows the light from a HeNe laser transmitted through a Ronchi ruling with 0.208 mm spacing with the full frequency spectrum being reflected on through the optical train to the CCD. The image on the right shows the same situation with the 2nd harmonic of the Fourier Transform being reflected and all other frequency components being diverted. The lower order components are easily observable in this situation. A more detailed description of the Fourier Transform laboratory experiment can be found in appendix B.

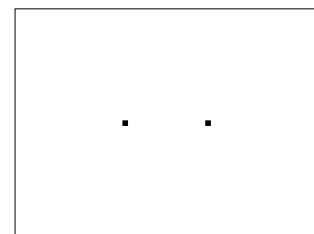


FIGURE 3: THE PATTERN USED TO REFLECT ONLY THE SECOND HARMONIC COMPONENTS OF THE FOURIER TRANSFORM OF A RONCHI RULING.

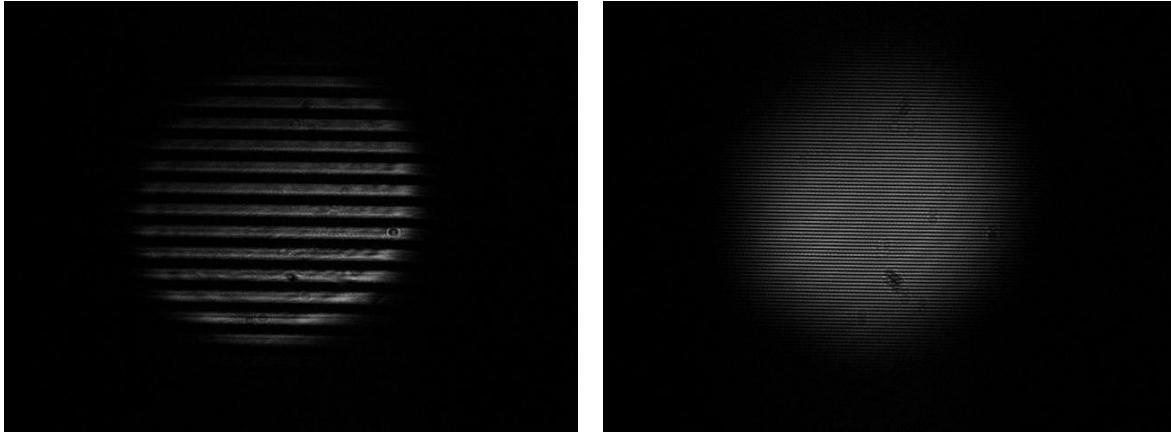


FIGURE 4: THE LEFT IMAGE SHOWS A FULL TRANSMISSION OF A LASER BEAM SHOWN THROUGH A RONCHI RULING WITH 0.208 MM SPACING. THE RIGHT IMAGE SHOWS THE TRANSMISSION OF THE 2ND HARMONIC OF THE SAME IMAGE.

A Fresnel Zone Plate (FZP) is traditionally a pattern that modifies the transmission of the zones of light wave-fronts. As a wave propagates the portion of the wave that is in phase is called a wave-front. A short distance behind that is another wave-front that is out of phase with the previous one. As the series of wave fronts propagate outward from the source they form a repeating set of spheres that meet a flat plane with a pattern that is a series of circular annuli that have a width equal to the half wavelength of the light or phase difference. A FZP blocks the out of phase components, passing components that are all in phase, to produce a more intense beam of light. The image in figure 4 shows an image of the focal point of a FZP that is designed using the FresnelZonePlate class in the bitmap_functions module to have a 0.6 m focal point. The image on the right is taken with the CCD placed at the focal point and the image on the left is taken at about 1/7 of the focal length. A detailed description of the Fresnel Zone Plate laboratory experiment is supplied in Appendix B.

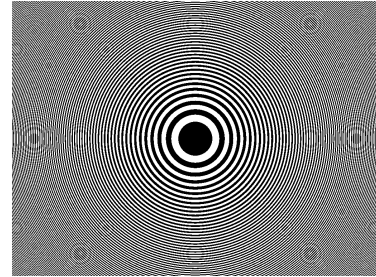


FIGURE 5: THE PATTERN THAT IS USED TO PRODUCE A FZP WITH $F=0.6$ M.

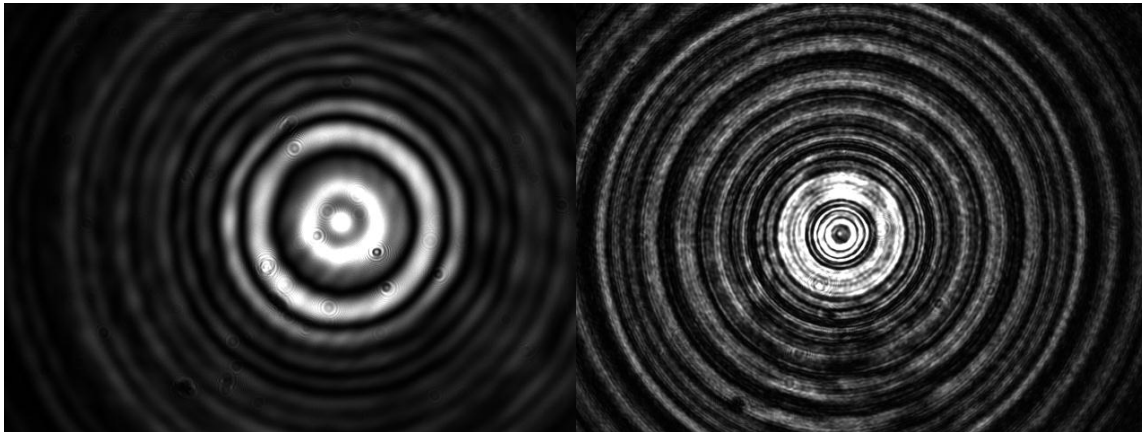


FIGURE 6: THE LEFT IMAGE WAS OBTAINED BY PLACING THE CCD AT 0.6 M. THE RIGHT IMAGE WAS TAKEN BY PLACING THE CCD AT ABOUT 0.08 M. THE FOCAL LENGTH OF THE FRESNEL ZONE PLATE IN THIS SETUP IS 0.6 M.

Coronagraphs

Bernard Lyot produced the concept of a coronagraph in 1930. It is usually a circular disk that blocks the intense light from a bright source, allowing for less bright objects to be observed. A coronagraph can be used to view the sun, during an eclipse the moon acts as a perfect coronagraph. Rayleigh scattering in the Earth's atmosphere can make it difficult to observe the Sun's corona. A polarizer used with a coronagraph can fix the problem, as shown in figure 4. The coronagraph produces scattering effects from the bright light source that can be problematic, especially when tracking distant objects.



FIGURE 7: AN IMAGE OF THE SUN TAKEN WITH A CORONAGRAPH TO BLOCK OUT THE SOLAR SPHERE AND A POLARIZER TO MAKE IT POSSIBLE TO DISTINGUISH THE SUNS CORONA FROM THE LIGHT IN THE EARTH'S ATMOSPHERE. [7] ON THE RIGHT IS A PATTERN THAT CAN BE USED TO ACHIEVE RESULTS COMPARABLE WITH THE CORONAGRAPH.

The application of the DMA apparatus in this sense is trivial, but produces significant results. This field will be demonstrated most apparently in the coming paragraphs. A coronagraph, in the scope of the DMA, is similar to the Lyot Stop mask described later, but with an opposing perspective. In this case the portion of the image being removed correlates with the bright light source. The experiments carried out use the idea of masking the light that passes through the optical system in a similar way as a traditional coronagraph.

Lyot Stop Mask

A Lyot Stop mask is another simple mask that can be accomplished using the DMA apparatus. In a coronagraph setup, the Lyot Stop is used to reduce the amount of star light that is scattered by the focal or image plane mask. This is similar to when a camera reduces the diameter of its iris on a bright day. The angular resolution of the telescope is reduced in using the Lyot Stop. More creative methods are being explored to image extra-solar planets that are based on this type of setup.

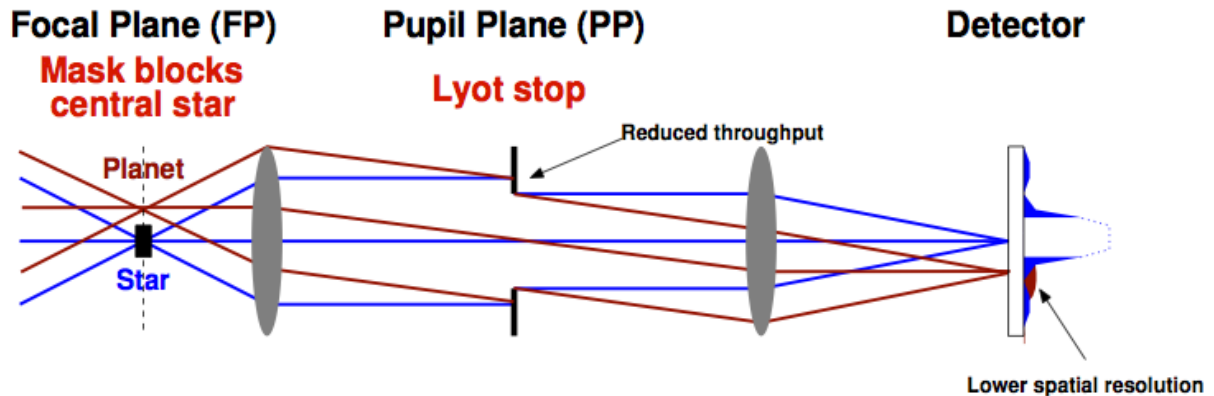


FIGURE 8: DIAGRAM REPRESENTING THE USE OF A LYOT STOP. A MASK PLACED INTO THE FOCAL PLANE OF A SYSTEM BLOCKS A BRIGHT STAR ALLOWING THE LIGHT OF AN EXOPLANET TO PASS THROUGH THE SYSTEM. THEN A LYOT STOP IS USED TO REDUCE THE TRANSMISSION OF LIGHT SCATTERED BY THE FOCAL PLANE MASK. [10]

In order to replicate this technique a simple circular pattern can be loaded into the DMA that only allows for the sought region to allow light to travel on to the CCD. A simple black circle is shown in figure 3 that demonstrates a Lyot Stop mask as it would be used in the DMA apparatus.

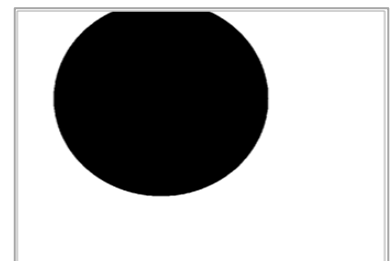


FIGURE 9: LYOT STOP MASK. CREATED USING SINGLE CIRCLE CLASS IN BITMAP FUNCTIONS MODULE.

Aperture Masking Interferometry

Aperture Masking Interferometry is a method of imaging developed by John E. Baldwin with the Cavendish Astrophysics Group at the University of Cambridge [3] and based on methods first suggested in 1970, by the French astronomer Antoine Labeyrie [1]. The method has also been implemented recently at the Keck Telescope in Maunaea Kea, Hawai'i. The Keck observatory uses a large diameter, ground based telescope with much of the aperture being blocked by the mask. Many small holes allow light to pass through and

produce a high resolution image that is diffraction limited, see figure 4. The diffraction limit is the fundamental maximum resolution that is possible with the optical system involved.

Aperture Masking Interferometry is a speckle imaging technique. Using Fourier analysis methods the information in a speckle pattern can be retrieved to produce a high-resolution image. The aperture is masked to form many small apertures and the light is recombined to form the image. Many short exposure images are then taken and stacked to create a single image. The mask pattern is designed to produce the best possible signal to noise ratio via bispectral analysis. Bispectral analysis allows most atmospheric noise to be removed from the image. This method simply covers the areas of the lens that does not pass the targets that are desired, therefore reducing the extra noise and increasing the ability to process out the unwanted light. The masks for the Keck telescope are made from aluminum sheet metal and are designed and cut for specific targets. The masks can take 10 minutes to change and are high above the primary mirror.

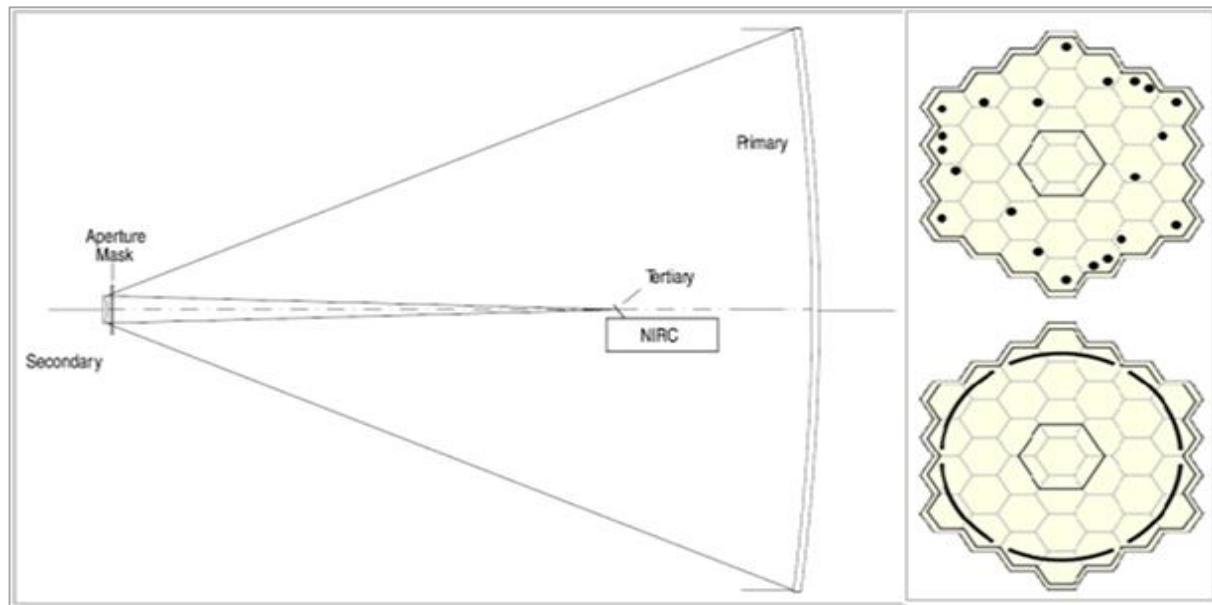


FIGURE 10: DIAGRAM OF APERTURE MASK POSITION FOR THE KECK TELESCOPE [5]. THE MASK IS PLACED NEAR THE SECONDARY MIRROR AND BLOCKS LIGHT AS IT PASSES IN BOTH DIRECTIONS. THE IMAGES ARE THEN CAPTURED WITH A NEAR INFRARED CAMERA (NIRC). [5] ON THE RIGHT ARE TWO MASKS USED AT KECK AND THEIR PROJECTED LAYOUT ON THE PRIMARY MIRROR AS THICK BLACK MARKINGS.

In this case the DMA apparatus is valuable because of its ability to rapidly change patterns. New patterns can be designed on the spot, removing the need for exact calculations of the targets positions. A mask of this sort might look like the mask provided in figure 5, the python programs have shown to be very versatile in their ability to create precise patterns for use in imaging. The drawbacks of course are the sizes of the instruments. The DMA apparatus does not have a resolving power that is in the same league as the Keck telescope. But, the basic concept is the same and can be applied easily.

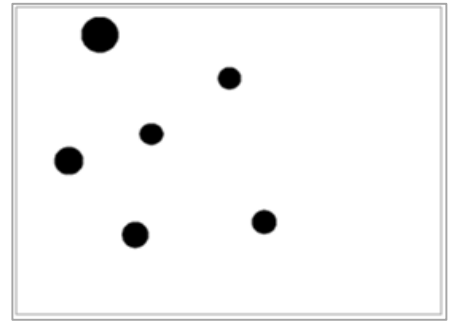


FIGURE 11: A REPRESENTATION OF A MASK FOR INTERFEROMETRY THAT COULD BE USED WITH THE DMA APPARATUS.

Experimental Results

In order to implement a coronagraph style optical mask a light source is targeted and imaged with the apparatus. The specific area of the DMA surface that contains the light source is then determined visually and a corresponding mask is produced. The mask is formed by creating a pattern that matches the shape of the source using the bitmap control module. For example, a circular pattern is produced by providing the coordinates of the center of the light source and a radius. When the mask is loaded onto the DMA, the mirrors corresponding to the light source divert the light away, and the required light is reflected through the apparatus.

E1. “Lab Stars” Experiment

Here the star simulation consists of 3 holes pierced into a piece of black cardboard. The center hole is about 0.5 cm from each of the two larger holes. Each of the larger holes are about the size of a pin point, and the small hole is about 1/3 of the large. There is a bright LED behind the cardboard producing uniform illumination. Figure 12 is an image of the cardboard at roughly 10 meters away with a mask consisting of two small rectangles. The separation is about 100 ± 20 arcseconds.

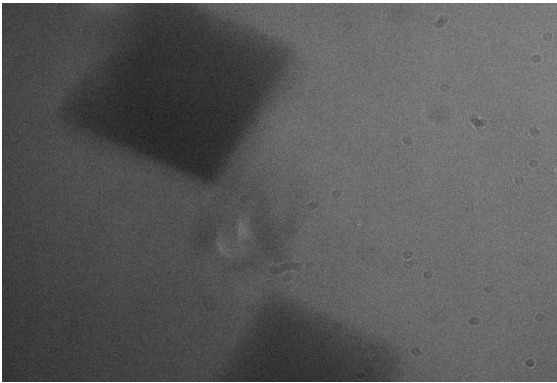


FIGURE 12: BLOCKED LAB STARS WITH 0.25 SECOND EXPOSURE TIME.

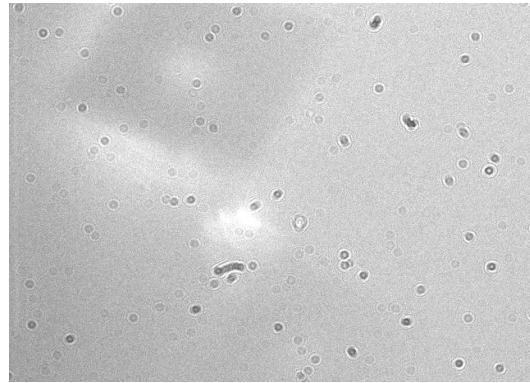


FIGURE 13: BLOCKED LAB STARS WITH 0.602 SECOND EXPOSURE TIME.

The second image in figure 13 is the same orientation taken with a longer exposure time.

Figures 14 and 15 are again the same images with no masks. As can be seen, the small hole is no longer discernible.

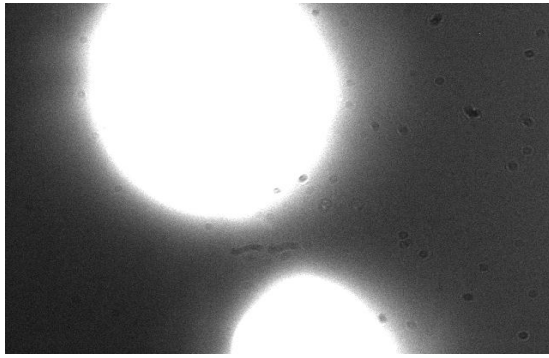


FIGURE 14: LAB STARS WITH 0.25 SECOND EXPOSURE TIME.

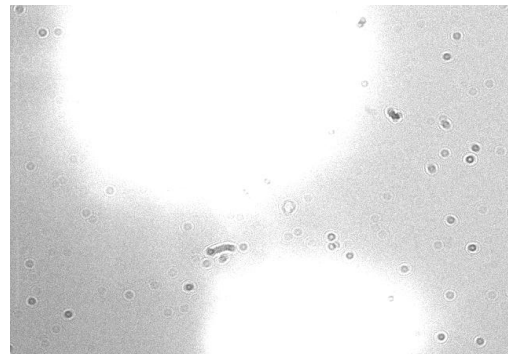


FIGURE 15: UNBLOCKED LAB STARS WITH 0.602 SECOND EXPOSURE TIME.

The exposure time of the CCD allowed for a special moment to occur and be captured. Figure 15 shows an image that takes place during the switching of the mask from “off” to “on”. The last image in this setup, figure 16, is again the same picture with an exposure time that matches that of the image in figures 12 and 14.

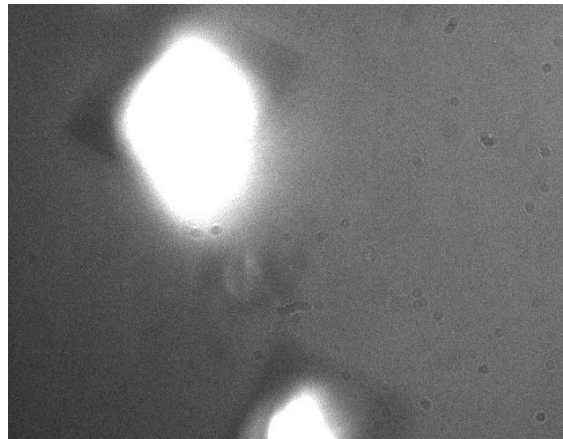


FIGURE 16: LAB STARS CAUGHT AT INSTANT OF BLOCKING TAKING PLACE WITH 0.25 SECOND EXPOSURE TIME.

In figures 14 and 15, two large bright light sources are visible with no clear evidence for a third light source half the distance between the two. The corresponding images in figures 12 and 13 reveal a small light source that is visible at the midpoint between the two large light sources.

E2. "Hall Star" Experiment



FIGURE 17: EXPERIMENTAL SIMULATION CONSISTING OF TWO LED'S. LEFT IMAGE IS SIDEVIEW WITH POWER ON. ON THE RIGHT IS AN ON AXIS IMAGE AS SEEN BY THE APPARATUS. THE RED LED IS SITUATED SO THAT IT IS UNDERNEATH AND PARTIALLY BLOCKED BY THE WHITE LED.

This experiment deals with a two LED star system simulation at a distance of about 40 meters. Above in figure 17, a close-up image of the simulation is provided for perspective. The bright LED is white with a 10 millimeter diameter and 28,500 millicandela intensity. The companion in this case is a red LED with a 3 millimeter diameter and a 0.8 millicandela intensity. In radiometric terms the difference in brightness of the white is about 5000 times the red. The red LED is located behind the white LED in such a way that it is only partially viewable by direct examination. Figures 18, 19, and 20 show the simulated binary star system imaged while the optical mask is in place. Various exposures are provided.

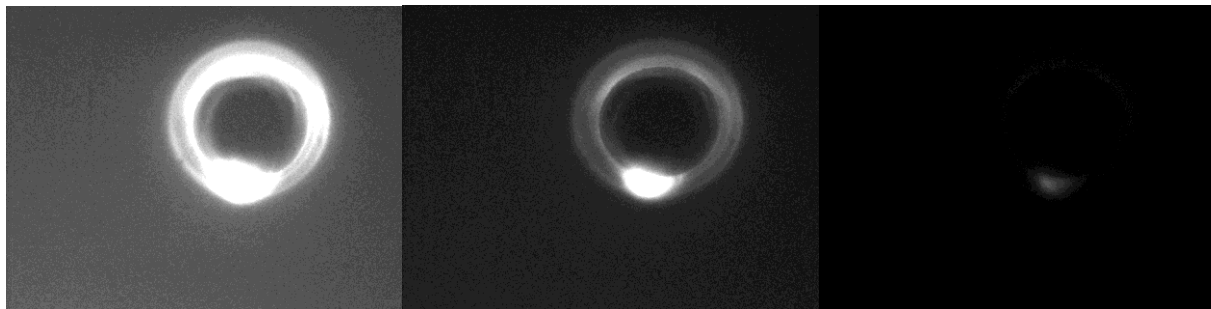


FIGURE 18: BLOCKED HALLSTAR AT 40 M WITH A 1/370 SECOND EXPOSURE.

FIGURE 19: BLOCKED HALLSTAR AT 40 M WITH 1/1000 SECOND EXPOSURE TIME.

FIGURE 20: BLOCKED HALLSTAR AT 40 M WITH 1/10000 SECOND EXPOSURE TIME.

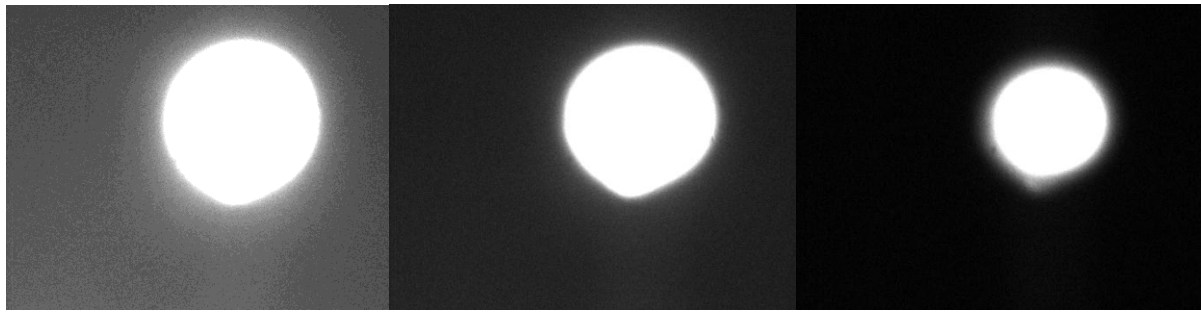
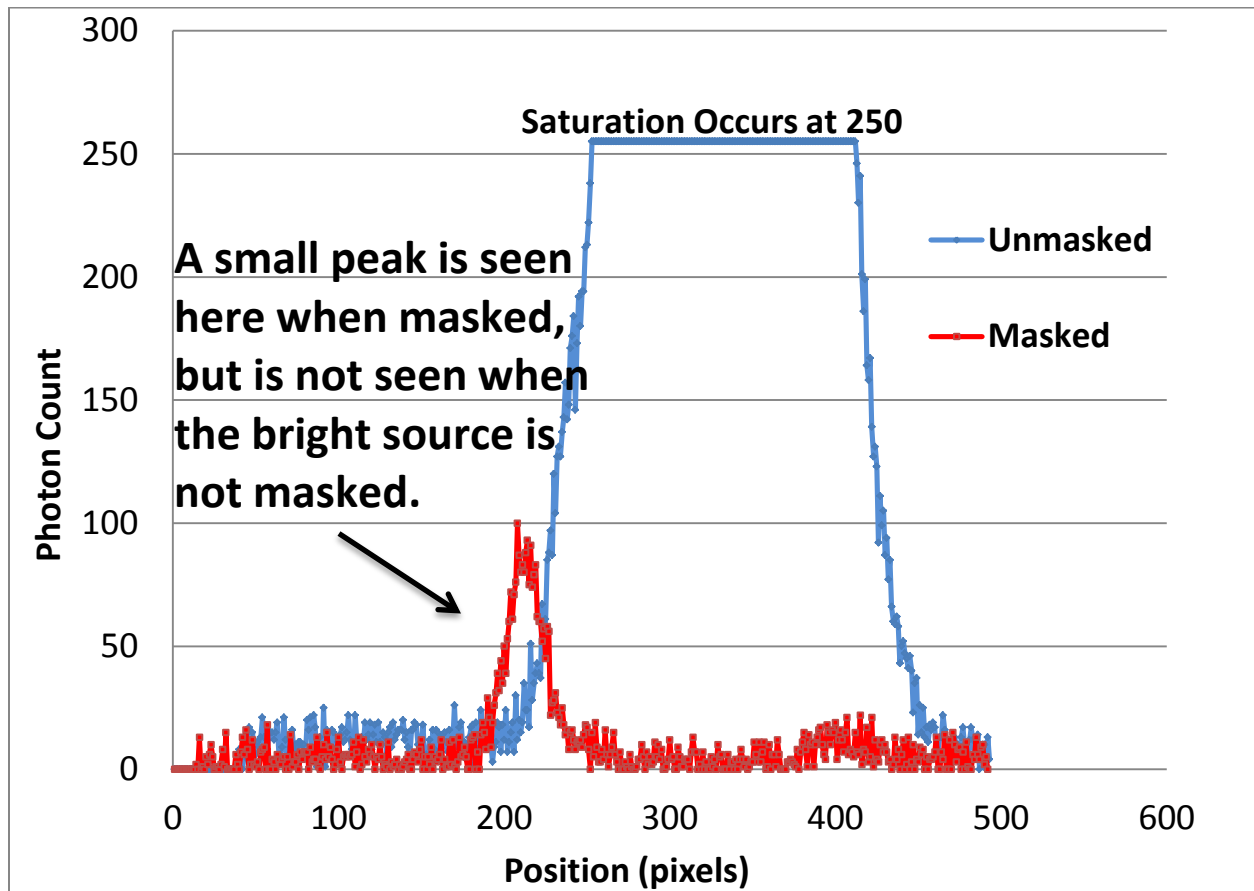


FIGURE 21: UNBLOCKED HALLSTAR AT 40 M WITH A 1/370 SECOND EXPOSURE.

FIGURE 22: UNBLOCKED HALLSTAR AT 40 M WITH 1/1000 SECOND EXPOSURE TIME.

FIGURE 23: UNBLOCKED HALLSTAR AT 40 M WITH 1/10000 SECOND EXPOSURE TIME.

The images in figures 21, 22, and 23 are the unblocked match to the previously discussed figures 18, 19, and 20. There is evidence of a companion in this set of data observed at the lower edge of the bright light source. The separation in this situation is 25 ± 5 arcseconds.



GRAPH 1: A GRAPHICAL REPRESENTATION OF THE PROFILES OF FIGURE 16 (RED), AND FIGURE 19 (BLUE). THE DATA SHOWS THE DIFFERENCE IN A MASKED IMAGE VS AN UNMASKED IMAGE. WITHOUT THE MASK THE CCD COULD NOT RECOGNIZE THE DIM COMPANION.

The plot of Photon Count vs. Position in Graph 1 shows the incident photons on the CCD for each of the images. In blue the counts show the profile of the light when the bright LED was not blocked out by the DMA, the red represents the counts when the DMA mask is

implemented. As can be seen, the unmasked image shows no trace of the dim companion even though the appearance of some type of light source is visible in the picture.

E3. “Monitor Stars” Experiment

The last data set obtained exploits the pixels of a monitor with a 0.216 millimeter dot pitch at a distance of about 13 meters. The angular separation for this experiment is 3.4 ± 0.1 arcseconds. This simulation consists of 5 pixels in a cross pattern with the right most pixel a dark grey dim companion. Figures 24 and 25 show the simulation with the 4 brighter pixels blocked out using two rectangles in an “L” shape at different exposure times. The images in figures 26 and 27 are the corresponding unblocked images.

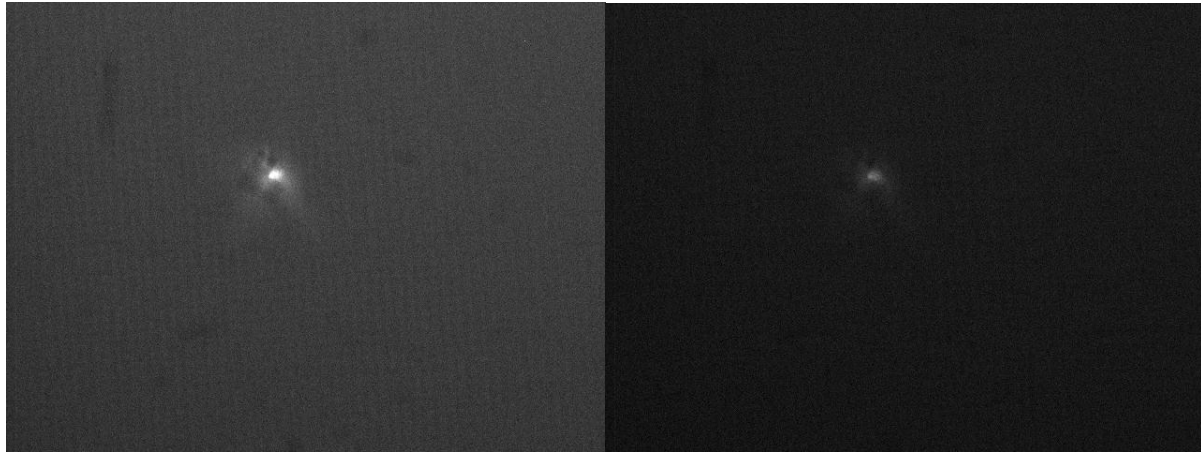


FIGURE 24: BLOCKED 4 PIXEL MAIN STAR WITH DARK GREY COMPANION 1/4 SECOND EXPOSURE TIME.

FIGURE 25: BLOCKED 4 PIXEL MAIN STAR WITH DARK GREY COMPANION 1/10 SECOND EXPOSURE TIME.

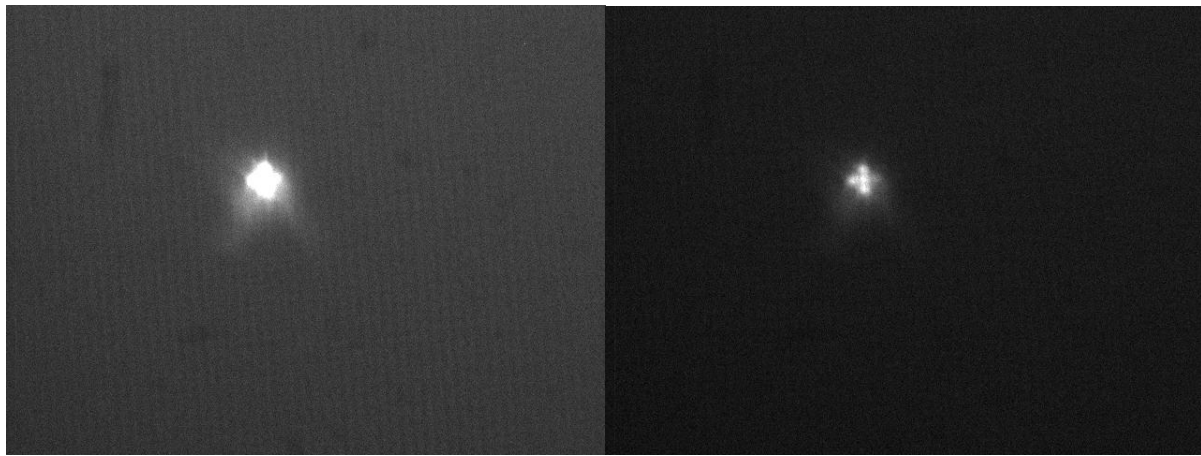


FIGURE 26: UNBLOCKED 4 PIXEL MAIN STAR WITH DARK GREY COMPANION 1/4 SECOND EXPOSURE TIME.

FIGURE 27: UNBLOCKED 4 PIXEL MAIN STAR WITH DARK GREY COMPANION 1/10 SECOND EXPOSURE TIME.

Discussion:

In the “Lab Star” experiment a first glimpse of the capabilities were captured. The outcome was a better understanding of the optical train and a procedure for aligning the apparatus with the target light source. The difficulties found by new astronomers were minimized because the light source was immobile. In an actual viewing session, the stars will be moving and the process will need to be faster in order to image such distant objects. However, the difficulty in using a mask that can adapt to the new situation proposes less problems than a stationary mask. The advantages of the DMA have already shown themselves to be useful.

The result obtained in the “Hall Star” experiment has provided further insight to the capabilities of the DMA apparatus. The circular symmetry of the LED allowed for a star-like object to be examined. Future considerations will include the diffraction effects of the mask, which present the opportunity to introduce another stop into the optical train to enhance the images. Currently the limits of the space and the ability to build a model that is small enough to properly reproduce a simulation that approaches the limits of the telescope hinder the ability to show the full limitations of the apparatus.

A third simulation was conducted using a computer monitor with the separation of each star being the pitch of the pixels. In this case the pixel pitch is 0.216 mm and measured at 13 m. The necessary distance for an accurate model of the Polaris system would be 333.5 m. This is a measurement that will be considered in the future. The data shows that, within the distances measured, the DMA can be used to block bright sources of light and allow improved imaging.

The OTF was calculated using the OTFfinder module of the Marina package. Using the convolution theorem the array that represents the Fourier transform of an image seen in figure 28 is divided by the array that represents the Fourier transform of the object in figure 29. And as discussed previously this produces the Fourier transform of the Point Spread Function or Optical Transfer Function. The OTF is provided in figure 30 as a 2 dimensional plot. This OTF is a preliminary assessment that only gives an approximation.



FIGURE 28: IMAGE TAKEN THROUGH DMA APPARATUS AND WILL BE USED AS THE IMAGE FOR THE OTF.

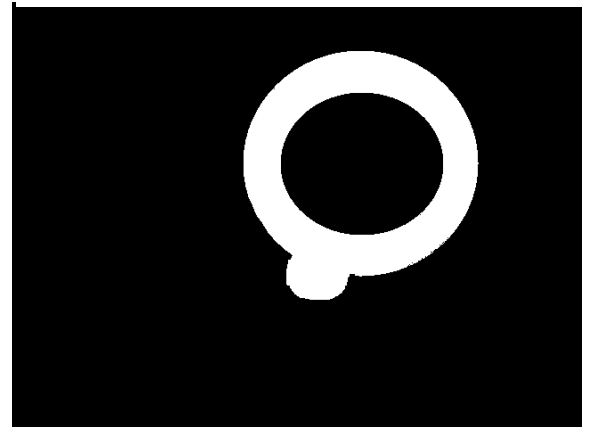
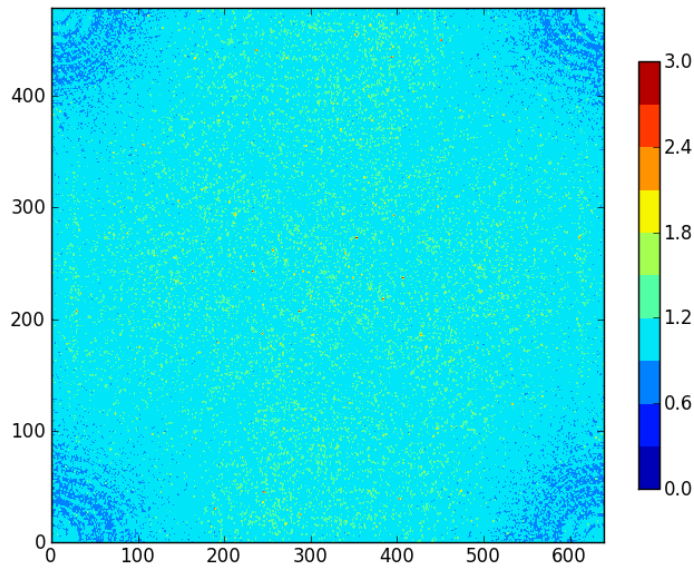


FIGURE 29: MONOCHROME PATTERN THAT WILL BE USED AS THE OBJECT FOR THE OTF.

\\DMA Research\Telescope_POE\Hallstar_blocked_132'er2''_hallstarmask1_R58_x
d D:\research\DMA Research\Telescope_POE\Hallstar_blocked_object.bmp



\\DMA Research\Telescope_POE\Hallstar_blocked_132'er2''_hallstarmask1_R58_x
d D:\research\DMA Research\Telescope_POE\Hallstar_blocked_object.bmp

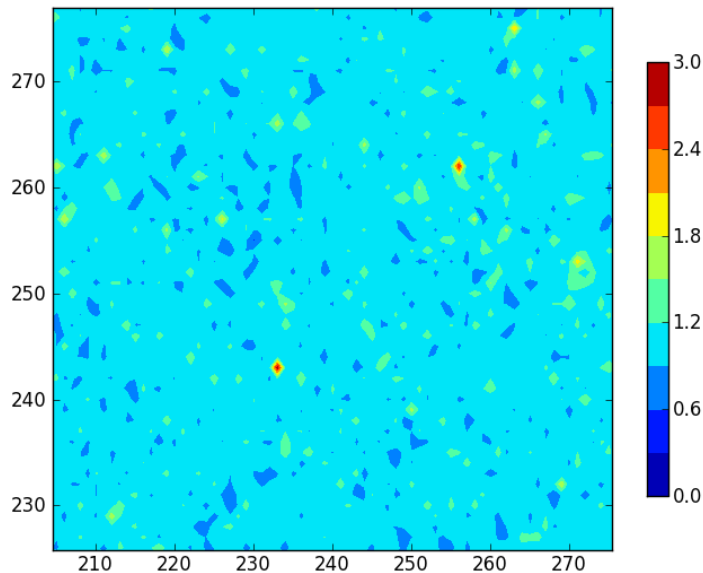


FIGURE 30: THE 2D PLOT ABOVE SHOWS THE ENTIRE OTF. THE LOWER PLOT SHOWS A CLOSE UP OF A PORTION OF THE FULL OTF TO EASIER PORTRAY THE DIFFERENCES IN THE VALUES.

The overall value of the OTF is a constant with a value of about 1. The corners show a trend in lower values, and there are various peaks in the central region, shown more apparently in the lower plot. A full diagnosis of the OTF is not complete at this time so speculative comments will be left to the reader.

Future works

Future work will include actual imaging of a multiple star system. A sampling of solar object imaging should be conducted including the less bright sections of the lunar surface and possibly the moons of Jupiter and Saturn and its rings. Further analysis of the optical transfer function after completion of the final structure should be done. The point spread functions for the full range of the field of view as observed by the CCD camera should be measured. The complete OTF will provide a tool for image processing in the future. The OTF can be divided out of the image to produce an improved image with the aberrations removed. Multiple images taken of a bright star with the incident light arriving at all points in the field of view would provide sufficient data for forming the OTF.

A number of experiments have been carried out using the DMA apparatus to recreate several optical procedures. Many of the principles that are taught in optics curricula can be replicated in a short amount of time. Once all of the data is created a demonstration of a principle can be loaded and observed.

Other work on software will include more tools being created for mask making and image analysis. In order to communicate directly to the DMA control board, software must be developed with C++, Active X, or some other programming language suitable.

Conclusion:

The DMD apparatus is capable of resolving images of binary star systems at significant resolutions. At the present time the apparatus appears to be capable of viewing nearby multiple star systems. The laboratory setup that is currently in use is not sufficient to prove that capability, but the results are not disappointing. The optical masks that have been implemented were successful in showing the basic abilities of the apparatus. The results of the "Lab Stars" experiment show that the system can be used to reveal small objects in a cluster of bright objects. The small center pinhole was not observable without the mask in place. When the mask was loaded onto the DMA the center hole was revealed at a resolution of 100 ± 20 arcseconds. The "Hall Stars" experiment used a greater distance than either of the other two experiments and showed that the intensity of a dominant bright light source could be drastically reduced using the DMA optical masking process. The dim companion in this case was almost totally hidden by the bright source and made obvious by the effects of the mask with an angular resolution of 25 ± 5 arcseconds. The "Monitor Stars" experiment showed that the system is still capable of functioning as hoped at greater resolutions. The angular resolution of 3.4 ± 0.1 arcseconds was achieved. The gathered data suggest that the DMA incorporated into the optical train is a very useful tool with still unseen possibilities.

References

- [1] Wikipedia contributors, "Speckle imaging," *Wikipedia, the free encyclopedia*. Wikimedia Foundation, Inc., 17-Mar-2012.
- [2A. Labeyrie, "Attainment of Diffraction Limited Resolution in Large Telescopes by Fourier Analysing Speckle Patterns in Star Images," *Astronomy and Astrophysics*, vol. 6, p. 85, May 1970.
- [3] Wikipedia contributors, "Aperture masking interferometry," *Wikipedia, the free encyclopedia*. Wikimedia Foundation, Inc., 21-Jul-2011.
- [4] J. E. Baldwin, C. A. Haniff, C. D. Mackay, and P. J. Warner, "Closure phase in high-resolution optical imaging," *Nature*, vol. 320, pp. 595–597, Apr. 1986.
- [5] P. G. Tuthill, J. D. Monnier, W. C. Danchi, E. H. Wishnow, and C. A. Haniff, "Michelson Interferometry with the Keck I Telescope," *Publications of the Astronomical Society of the Pacific*, vol. 112, no. 770, pp. 555–565, Apr. 2000.
- [6] "DLP System Optics" Application Report DLPA022, Texas Instruments, July 2010.
- [7] Wikipedia contributors, "Coronagraph," *Wikipedia, the free encyclopedia*. Wikimedia Foundation, Inc., 16-May-2012.
- [8] "Bitmap Format." [Online]. Available: <http://atlc.sourceforge.net/bmp.html>. [Accessed: 24-May-2012].
- [9] "LX200 Series - Specifications," *Meade Instruments - Telescopes, Solar Telescopes, Accessories, Telescopes by Meade*. [Online]. Available: <http://meade.com/lx200/specifications>. [Accessed: 30-May-2012].
- [10] "Robust Coronagraphy with an APP." [Online]. Available: <http://home.strw.leidenuniv.nl/~kenworthy/research/app/>. [Accessed: 04-Jun-2012].

Appendix A - Software

The DMA package is provided with software used to load data onto the DMA itself as an array. DLP Discovery 4100 Explorer Software reads bitmap files as 1's and 0's and relates that to the off and on position of each mirror in the array.

The custom software used for data creation was written in the Python programming language. An existing package called Marina was written as a cooperative development by Dr. William Hetherington and some of his students. The programs used for the DMA were additions to the Marina package. A graphical user interface (GUI), named bitmap_control was created as a control module for all other programs related to the DMA. The GUI accesses other programs to perform tasks as needed.

DMAScriptWriter was one of the first programs created to run in the DMA GUI. This program prompts the user for data files and writes a text file that can be interpreted by the Explorer software. This was an easy way to produce images at the beginning of the experiment and provides insight into how the data is loaded into the field programmable gate array (FPGA) and then to the DMA.

bitmap_functions is a module that holds smaller classes of functions. The program functions are mathematical functions in this case and they are used to create data sets in the form of an array. The parameters of the arrays are determined by yet another very versatile module called ParameterPicker. ParameterPicker is kind of the middle man when the GUI is prompted for a specific type of function. Each function in bitmap_functions has its own parameters, so ParameterPicker reads the types of data that are needed from the user and produces a GUI in the bitmap_control GUI that will allow for specific variables to be changed according to the needs of the user. It then enters those parameters into the function and the array is created.

show_mask is a small but very important part of all data set creation. It reads the arrays that are produced by bitmap_functions and formats them into a monochrome bitmap that the Explorer software can deliver to the DMA.

myModuleMaker and My_Functions are programs that can be used by a new user in the Marina package to create new GUI's and a new set of functions similar to bitmap_functions. The instructions for personalization are in the code to provide quick and easy access for new researchers.

The Explorer Software that came with the DMA package has the capability to load images onto the DMA. The image can be loaded in various methods such as single rows, single blocks, dual blocks, and quad blocks and global. There are 16 blocks of rows $768/16=48$ rows per block. The single row load does exactly as it sounds and loads a single row. The blocks load the blocks of rows and global loads the entire array of mirrors. The entire row must be loaded in order to change even one pixel. The Reset command changes the mirrors to the format that has been loaded. A common command for our use is the global load and reset command which loads an entire image and then resets the mirrors to portray the image on the DMA.

Most of the programs discussed are provided in the following pages. If there is an interest in any of the others you can contact Dr. William Hetherington in order to find a copy of the full package.

Python Code

bitmap_control

```
import sys, time, struct, os, wx
import numpy as np
import numpy.linalg as lin
import os.path
import wx.lib.dialogs as wd # Not the same as importing as a system module for some unknown reason.
import shutil              # used for saving files
import winsound

#required_modules = ['DMAgrid', 'DMAScriptWriter', 'wx.lib.dialogs']

# Description -----

# Module for control of the DMA System

#-----

class bitmap_control():

    def __init__(self, dock):
        self.dock = dock
        self.wx = dock.wx
        self.version = '1'
        self.name = 'bitmap_control'
        self.temp_file = 'binary.bmp'
        self.display_name = 'Bitmap Control'
        self.openftn_file = ""
        self.openftn_file_ext = "*.py#|*.csv|*.dat|*.txt|*" # csv files (*.csv)|*.csv | # txt files (*.txt)|*.txt | dat files (*.dat)|*.dat |
        self.openftn_directory = 'C:\Documents and Settings\ph411\Desktop\DMA Research\marina\app\DMA\modules\Functions'
        self.save_file = 'binary.bmp'
        self.save_directory = 'C:\Documents and Settings\ph411\Desktop\DMA Research\marina\app\DMA\modules\maskbitmaps'
        self.content = 'This module provides an interface that makes it possible to create bitmaps that are structured for display on the DMA.'
        self.about = "Module: Bitmap Control\n" \
            + "Purpose: control....bitmaps\n" \
            + "Authors:lt W. Hetherington <- framework, modified by Shawn Gilliam.\n" \
            + "Created:lt 2011\n" \
            + "Copyright:lt (c) 2011\n" \
            + "License:lt No restrictions\n"
        self.help = "C:/Documents and Settings/ph411/Desktop/DMA Research/marina/app/DMA/help/bitmap_controltab_help.txt"

        #The following is a list of keys and explanations for the 'controls' dictionary
        #keys must match the example
        #name' - value is name you choose that will appear at top of tab
        #order' - The position of the control, beginning at 0, in the order in which you desire the controls to
        appear.

        #type' - Value must be named, 'button','input_and_button'(text input/button pair), 'text_area',
        'static_text'

        #label' - Value is the name that will appear on the button
        #function' - Reference to the function to which the button event will respond.

        self.controls = {}
        self.ctl_count = 0
        self.DefineControl('setup', {'format':'control_2', 'tab':'yes', 'rows':4, 'columns':4})
        self.DefineControl('function_importer', {'type':'button', 'label':'Open Function Module', 'function':self.FunctionImport})
        self.DefineControl('mask_selection', {'type':'combo_box', 'dimensions':[200,100], 'label':'Create Mask', 'function':self.MaskSelection})
        self.DefineControl('save_button', {'type':'button', 'label':'Save File', 'function':self.OnSave})
        self.DefineControl('about_button', {'type':'button', 'label':'About This Module', 'function':self.OnAbout})
        self.DefineControl('help_button', {'type':'button', 'label':'Help', 'function':self.OnHelp})
        self.DefineControl('grid', {'type':'button', 'label':'DMA Grid Background', 'function':self.pixelmask})
        self.DefineControl('scripter', {'type':'button', 'label':'That old time script writer', 'function':self.scriptwriter})
        self.DefineControl('2d_iFT', {'type':'button', 'label':'Fast Fourier Transform', 'function':self.OnFFT})
        self.DefineControl('OFT', {'type':'button', 'label':'Optical Transfer Function', 'function':self.OnOTF})
        self.DefineControl('2d_iFT', {'type':'button', 'label':'Inverse FFT', 'function':self.OnIFFT})
        self.DefineControl('ImgInt', {'type':'button', 'label':'Image Intensity', 'function':self.ImgInt})
        self.DefineControl('LocalMaxFinder', {'type':'button', 'label':'Local Maximum Locator', 'function':self.LocalMaxLocate})

    def DefineControl(self, ctl_name, c):
        if ctl_name.lower() != 'setup':
            c['order'] = self.ctl_count
            self.ctl_count += 1
        self.controls[ctl_name] = c

    def PostCreationConfigure(self):
        # Label some system or tabbed modules which will be used.
```

```

self.wd = self.dock.system_modules['wx.lib.dialogs']    # This is a different module?
self.tab = self.dock.tabs[self.name]
self.wd = wd
self.bf = self.dock.system_modules['bitmap_functions']
self.sm = self.dock.system_modules['show_mask']
self.pp = self.dock.system_modules['ParameterPicker']
self.fft = self.dock.system_modules['TwoDFFTmodule']
self.otf = self.dock.system_modules['OTFfinder']
self.ifft = self.dock.system_modules['TwoDFFTmodule']
self.imgint = self.dock.system_modules['ImageIntensity']
self.lmf = self.dock.system_modules['LocalMaxFinder']
self.np = self.dock.system_modules['numpy']
#self.rpf = self.dock.system_modules['response_plot_files']
self.x = self.np.array([])
self.y = []
self.filein = "    # Will be replaced by a filein object.

# Define the control boxes AFTER the controls have been created.
self.mask_selection_box = self.dock.tabs[self.name].controls['mask_selection']
#self.function_finder_box = self.dock.tabs[self.name].controls['function_finder']    # The arrays are listed here.

#####Section for creating list of functions in the mask selection box#####
self.funk = self.bf.ListFunctions(self.bf)    # This is a list of function objects, ie instances of each function class.
for f in self.funk :
    self.mask_selection_box.Append(f.name)

def FunctionImport(self, event) :
    self.error = "
    self.mask_selection_box.Clear()
    d = self.wx.FileDialog(self.tab.frame, 'Choose a function Module', self.openftn_directory, self.openftn_file, self.openftn_file_ext, self.wx.FD_OPEN,
self.wx.DefaultPosition)
    if d.ShowModal() != self.wx.ID_OK :
        #self.Notify('Read error: ')
        return self.error
    self.openftn_directory = d.GetDirectory()
    self.openftn_file = d.GetFilename()
    self.openftn_path = d.GetPath()
    module_name = self.openftn_file
    stuff = module_name.split('.')
    if (stuff[-1].strip() != 'py') or (len(stuff) > 2) :
        print 'Erroneous module name : ' + module_name
        return
    self.functions = self.dock.system_modules[stuff[0]]
    self.funk = self.functions.ListFunctions(self.functions)
    for f in self.funk :
        self.mask_selection_box.Append(f.name)
    return self.error

def MaskSelection(self, event) :
    # Function is chosen by the user from a combo box
    f_choice = self.mask_selection_box.GetCurrentSelection()    # Function is chosen by the user from a combo box
    f = self.funk[f_choice]
    xpix = 1024
    ypix = 768
    contour_value = 0.05    # This is entered by the user in a text box.
    self.show_image = True    # show_image is set to True or False
    self.parameters = f.parameters
    pd = self.pp.ParameterDialog(None, -1, 'Input ' + self.funk[f_choice].name + ' Parameters', self)
    pd.ShowModal()
    pd.Destroy()
    show_image = self.show_image
    parameters = self.parameters
    if show_image :
        error, header, dat, elapsed_time = self.sm.CreateBMP(self.funk[f_choice].func, parameters, xpix, ypix, contour_value)
        print elapsed_time
        if error :
            print 'Abort program.'
            return
        print 'Binary size : ' + str(len(dat))
        try :
            f = open(self.temp_file, 'wb') # the b for binary is needed only on windows systems because of the end of line character issue.
            f.write(header)
            f.write(dat)
            f.close()
        except :
            print 'File ' + self.temp_file + ' could not be written.'
            return
        sb = self.sm.ShowImage(self.funk[f_choice].name, self.temp_file, image_type='bmp')

def OnFFT(self, event):
    pf = self.fft.TwoDPowerSpectrum(None, -1, "", self)
    #pd.ShowModal()
    #pd.Destroy()

```

```

def OnOTF(self, event):
    pf = self.otf.OTFdialog(None, -1, "", self)

def OniFFT(self, event):
    pg = self.iffi.TwoDInvPowerSpectrum(None, -1, "", self)

def ImgInt(self, event):
    pf = self.imgint.ImageIntensityShow(None, -1, "", self)

def LocalMaxLocate(self, event):
    pf = self.lmf.LocalMaxFinder(None, -1, "", self)

def pixelmask(self, event) :
    path_to_module = 'modules/'
    module_name = 'DMAgrid.py'
    stuff = module_name.split('.')
    if (stuff[-1].strip() != 'py') or (len(stuff) > 2) :
        print 'Erroneous module name .' + module_name
        return
    try :
        showgrid = __import__(stuff[0])
    except :
        print 'Import error .' + stuff[0]
        return
    # The idea with this module was to stream the cod camera as the background and then as you click each pixel in the
    # screen it would turn off the corresponding mirror. Use pygame library if you want to continue work.

#def scriptwriter(self, event) :
#    path_to_module = 'modules/'
#    module_name = 'DMAScriptWriter.py'
#    stuff = module_name.split('.')
#    if (stuff[-1].strip() != 'py') or (len(stuff) > 2) :
#        print 'Erroneous module name .' + module_name
#        return
#    try :
#        showscrip = __import__(stuff[0])      # Remove the '.py'
#    except :
#        print 'Import error .' + stuff[0]
#        return
#    # at a later time I will introduce wx_practice to create a manipulatable script

def OnSave(self, event) :
    d = self.wx.FileDialog(self.tab.frame, 'Save File', self.save_directory, self.save_file, "", self.wx.FD_SAVE | self.wx.FD_OVERWRITE_PROMPT, self.wx.DefaultPosition)
    if d.ShowModal() == self.wx.ID_OK : # Show it.
        self.save_directory = d.GetDirectory()
        self.save_file = d.GetFilename()
        self.save_path = d.GetPath()
        shutil.copyfile(self.temp_file, self.save_path)
    d.Destroy() # Destroy it when finished.

def OnAbout(self, event):
    #print self.name
    #print self.dock.tabs[self.name].tab
    d = self.wx.MessageDialog( self.dock.tabs[self.name].tab, self.about, "About Bitmap Control", self.wx.ICON_INFORMATION) # Create a message dialog
box
    d.ShowModal() # Show it.
    d.Destroy() # Destroy it when finished.

def OnHelp(self, event):
    f = open(self.help)
    help_me = f.read()
    f.close()
    d = self.wx.ScrolledMessageDialog(self.dock.tabs[self.name].tab, help_me, caption="Laser light can be hazardous - Please wear Sunglasses!",
pos=(100,100), size=(600,500))#, style=self.wx.HSCROLL)
    d.ShowModal() # Show it.
    d.Destroy() # Destroy it when finished.

# The setup routine must be in each application module. -----

def setup(dock) :
    # Create the module object associated with a tab of a similar name.
    object = bitmap_control(dock)      # The module object will be aware of dock.
    return object

# Optional test routines -----

def WrongMod() :
    print 'This is not the module you want. Execute control.py'

#-----

# Execute
if __name__ == '__main__':

```

```
WrongMod()
```

show_mask

Test module for an application which generates a bmp.

```
import sys, time, struct
import numpy as np
import numpy.linalg as lin
import wx
```

```
class ShowImage(wx.App):
    def __init__(self, image_name, image_file, image_type='any'):
        self.error = ""
        image_type = image_type.strip().lower()
        if image_type == 'bmp':    imgtyp = wx.BITMAP_TYPE_BMP          # Load a Windows bitmap file.
        elif image_type == 'gif':  imgtyp = wx.BITMAP_TYPE_GIF        # Load a GIF bitmap file.
        elif image_type == 'jpg':  imgtyp = wx.BITMAP_TYPE_JPEG       # Load a JPEG bitmap file.
        elif image_type == 'png':  imgtyp = wx.BITMAP_TYPE_PNG        # Load a PNG bitmap file.
        elif image_type == 'pcx':  imgtyp = wx.BITMAP_TYPE_PCX        # Load a PCX bitmap file.
        elif image_type == 'pnm':  imgtyp = wx.BITMAP_TYPE_PNM        # Load a PNM bitmap file.
        elif image_type == 'tif':  imgtyp = wx.BITMAP_TYPE_TIF        # Load a TIFF bitmap file.
        elif image_type == 'xpm':  imgtyp = wx.BITMAP_TYPE_XPM        # Load a XPM bitmap file.
        elif image_type == 'ico':  imgtyp = wx.BITMAP_TYPE_ICO        # Load a Windows icon file (ICO).
        elif image_type == 'cur':  imgtyp = wx.BITMAP_TYPE_CUR        # Load a Windows cursor file (CUR).
        elif image_type == 'ani':  imgtyp = wx.BITMAP_TYPE_ANI        # Load a Windows animated cursor file (ANI).
        elif image_type == 'any':  imgtyp = wx.BITMAP_TYPE_ANY        # Will try to autodetect the format.
        else:
            self.error = 'Unknown file type : ' + image_type
            print self.error
            return
        wx.App.__init__(self)
        self.frame = wx.Frame(None, title=image_name)
        self.panel = wx.Panel(self.frame)
        wxImg = wx.EmptyBitmap( 1, 1 )    # Create a bitmap container.
        wxImg.LoadFile(image_file, imgtyp)    # Load a bitmap into the container.
        self.imageCtrl = wx.StaticBitmap(self.panel, id=-1, bitmap=wxImg)
        self.mainSizer = wx.BoxSizer(wx.VERTICAL)
        self.mainSizer.Add(self.imageCtrl, 0, wx.ALL, 5)
        self.panel.SetSizer(self.mainSizer)
        self.mainSizer.Fit(self.frame)
        self.panel.SetSizer(self.mainSizer)
        self.panel.Layout()
        self.frame.Show()
        self.MainLoop()
        return

    def OnPlotChannels(self, event):
        if len(self.chans) == 0:
            self.OnError("Nothing to plot. Channel list is empty.")
            return
        b = self.dock.system_modules['notebook_plots_base']
        g = self.dock.system_modules['notebook_plots_graph']
        f = b.PlotFrame(title='Channels versus Time', size=(800,800))
        plist = []
        legends = []
        for i in range(len(self.chans)):
            if int(self.chans[i]) == 1:
                xunit = self.active_scope_obj.ch1.xunit
                yunit = self.active_scope_obj.ch1.yunit
            else:
                xunit = self.active_scope_obj.ch2.xunit
                yunit = self.active_scope_obj.ch2.yunit
            yunit = yunit.strip()[0]
            chvt = g.Graph('plot', [[self.x, self.ys[i]]], symbols=['b','g'], grid=False, title = 'Ch '+self.chans[i]+' vs Time', titlesize=24,\
                background='w', xlabel=r'Time ('+xunit+')', ylabel=r'Signal ('+yunit+')', labelsz=18, ticklabelsize=14, customize=None)#,
            legend=[r'1', r'2'])
            g.DefineTab(f, [chvt], name='Channel '+self.chans[i]+' vs Time')
            plist.append((self.x, self.ys[i]))
            legends.append(self.chans[i])
        allchvt = g.Graph('plot', plist, symbols=["", "", ""], grid=False, title = 'All Channels vs Time', background='w', \
            xlabel=r'Time ('+xunit+')', ylabel=r'Signal ('+yunit+')', labelsz=18, ticklabelsize=14, customize=None)#, legend=[r'1','2'])
        g.DefineTab(f, [allchvt], name='All Channels vs Time')
        f.Show()

def CreateBMP(f, parameters, xpix, ypix, contour_value):
    # Create a bit map as a binary block.
    # Sources:
    # Examples : http://www.doughellmann.com/PyMOTW/struct/
    # Reference: http://docs.python.org/library/struct.html
    error = "
```



```

# Determine the number of bytes in the monochrome image, for which 1 bit represents one pixel.
image_pix = xpix * ypix
image_bytes = image_pix/8
image_high_int16 = image_bytes/65536 # The high 16-bit word of the 32-bit number
image_low_int16 = image_bytes - image_high_int16*65536 # The low 16-bit word of the 32-bit number
total_bytes = image_bytes + 62
total_high_int16 = total_bytes/65536 # The high 16-bit word of the 32-bit number
total_low_int16 = total_bytes - total_high_int16*65536 # The low 16-bit word of the 32-bit number
# Monochrome bitmap header. See http://atlc.sourceforge.net/bmp.html, http://paulbourke.net/dataformats/bitmaps/
a = [19778, total_low_int16, total_high_int16, 0, 0, 62, 0, 40, 0, xpix, 0, ypix, 0, 1, 1, 0, 0, image_low_int16, image_high_int16, 3937, 0, 3937, 0, 2, 0, 2, 0, 65535, 255, 0,
0]

alen = len(a)
X = np.arange(xpix)
Y = np.arange(ypix)
error, func, elapsed_time = f(X, Y, parameters)
if error :
    print 'Abort process.'
    return error, 0, 0, elapsed_time

d = []
b=parameters['FFT']['value']
if b == 1:
    for v in func:
        d.append(v)
    darray=np.array(d)
    darray=np.reshape(darray, (768, 1024))
    darrayfft=np.fft.fft2(darray)
    d=np.reshape(darrayfft, (786432, 1))
    mask=[]
    i=0
    while i < len(d):
        g=d[i]
        if g > 800 :
            mask.append(1)
        else:
            mask.append(0)
        i=i+1
    c = []
    n = 0
    while n < image_pix :
        z = 0
        for i in range(16) :
            z += mask[n + i] * 2**(15-i) # Assuming big-endian order in the bmp.
        c.append(z)
        n += 16
    else:
        mask=[]
        for v in func :
            if v > 0.05 :
                mask.append(1)
                #row[x] = 1
            else :
                mask.append(0)
        c = []
        n = 0
        while n < image_pix :
            z = 0
            for i in range(16) :
                z += mask[n + i] * 2**(15-i) # Assuming big-endian order in the bmp.
            c.append(z)
            n += 16

    numints = len(c)
    header = a # header = np.array(a, dtype=np.uint16) # unsigned short 16-bit (2-byte) integers..
    s = struct.Struct('<'+str(alen)+'H') # short unsigned integers little endian
    e = s.pack(*header) # The * is a very useful function. If a = [1, 2, 3], then f(*a) = f(1, 2, 3) not f([1, 2, 3]).
    #print len(e)
    s = struct.Struct('>'+str(numints)+'H') # short unsigned integers big endian
    dat = s.pack(*c) # The * is a very useful function. If a = [1, 2, 3], then f(*a) = f(1, 2, 3) not f([1, 2, 3]).
    return error, e, dat, elapsed_time

#-----
if __name__ == '__main__':
    Test()

```

function_bitmap

Test module for an application which generates a bmp.

```

import sys, time, struct
import numpy as np
import numpy.linalg as lin
import wx

def Test():
    # Create a monochrome bitmap from a function and a contour value.
    path_to_module = "          #functions/"
    module_name = 'bitmap_functions.py'
    if len(path_to_module.strip()) > 0 :
        sys.path.append(path_to_module)
    stuff = module_name.split('.')
    if (stuff[-1].strip() != 'py') or (len(stuff) > 2) :
        print 'Erroneous module name :.' + module_name
        return
    try :
        funcs = __import__(stuff[0])          # Remove the '.py'
    except :
        print 'Import error : ' + stuff[0]
        return
    fl = funcs.ListFunctions(funcs)           # This is a list of function objects, ie instances of each function class.
    if len(fl) == 0 :
        print 'Error: No function classes found in ' + stuff[0]
        return
    # This loop is useful only for debugging.
    for i in range(len(fl)) :
        f = fl[i]
        print str(i) + ' ' + f.name + ' : ' + f.description
        print f.required_parameters
        xpix = 1024      # Any reasonable value.
        ypix = 768       # Any reasonable value.
        contour_value = 0.05      # This is entered by the user in a text box.
        f_choice = 3      # Function is chosen by the user from a combo box
        if fl[f_choice].name == 'Periodic Ellipses Solid' :
            parameters = {'k' : 2.0*np.pi/200.0, 'yo' : ypix/2}          # This is entered by the user in a multi-line text box.
        elif fl[f_choice].name == 'Circular Sinc' :
            parameters = {'k' : 2.0*np.pi/200.0, 'xo' : xpix/2, 'yo' : ypix/2}          # This is entered by the user in a multi-line text box.
        elif fl[f_choice].name == 'Fresnel Zone Plate' :
            parameters = {'f' : 1.0, 'lambda' : 6.33e-07, 'xo' : xpix/2, 'yo' : ypix/2}          # This is entered by the user in a multi-line text box.
        elif fl[f_choice].name == 'Fresnel Zone Plate Cylindrical' :
            parameters = {'f' : 1.0, 'lambda' : 6.33e-07, 'xo' : xpix/2}          # This is entered by the user in a multi-line text box.
        else :
            print 'Function not found.'
            return
    # These two lines display a busy statement.
    #busy = self.wx.BusyInfo("Image creation in progress ... ", self.tab.frame)
    #self.wx.Yield()          # necessary, otherwise busyinfo display is empty on Linux.
    error, header, dat, elapsed_time = funcs.CreateBMP(fl[f_choice].func, parameters, xpix, ypix, contour_value)
    if error :
        print 'Abort program.'
        return
    print elapsed_time
    print 'Binary size : ' + str(len(dat))
    temp_file = 'binary.bmp'          # This is a temporary file. Use a save-file dialog to move this to your location.
    try :
        f = open(temp_file, 'wb')      # the b for binary is needed only on windows systems because of the end of line character issue.
        f.write(header)
        f.write(dat)
        f.close()
    except :
        print 'File ' + temp_file + ' could not be written.'
        return
    show_image = True          # show_image is set to True or False using a selection box control.
    if show_image :
        ShowImage(fl[f_choice].name, temp_file, image_type='bmp')
    # The save-file dialog goes here.

class ShowImage(wx.App):
    def __init__(self, image_name, image_file, image_type='any'):
        self.error = "
        image_type = image_type.strip().lower()
        if image_type == 'bmp' :      imgtyp = wx.BITMAP_TYPE_BMP          # Load a Windows bitmap file.
        elif image_type == 'gif' :      imgtyp = wx.BITMAP_TYPE_GIF          # Load a GIF bitmap file.
        elif image_type == 'jpg' :      imgtyp = wx.BITMAP_TYPE_JPEG          # Load a JPEG bitmap file.
        elif image_type == 'png' :      imgtyp = wx.BITMAP_TYPE_PNG          # Load a PNG bitmap file.
        elif image_type == 'pcx' :      imgtyp = wx.BITMAP_TYPE_PCX          # Load a PCX bitmap file.
        elif image_type == 'pnm' :      imgtyp = wx.BITMAP_TYPE_PNM          # Load a PNM bitmap file.
        elif image_type == 'tif' :      imgtyp = wx.BITMAP_TYPE_TIF          # Load a TIFF bitmap file.
        elif image_type == 'xpm' :      imgtyp = wx.BITMAP_TYPE_XPM          # Load a XPM bitmap file.
        elif image_type == 'ico' :      imgtyp = wx.BITMAP_TYPE_ICO          # Load a Windows icon file (ICO).
        elif image_type == 'cur' :      imgtyp = wx.BITMAP_TYPE_CUR          # Load a Windows cursor file (CUR).
        elif image_type == 'ani' :      imgtyp = wx.BITMAP_TYPE_ANI          # Load a Windows animated cursor file (ANI).
        elif image_type == 'any' :      imgtyp = wx.BITMAP_TYPE_ANY          # Will try to autodetect the format.
        else :
            self.error = 'Unknown file type :.' + image_type
            print self.error

```

```

        return
wx.App.__init__(self)
self.frame = wx.Frame(None, title=image_name)
self.panel = wx.Panel(self.frame)
        wxImg = wx.EmptyBitmap( 1, 1 )    # Create a bitmap container.
        wxImg.LoadFile(image_file, imgtyp)    # Load a bitmap into the container.
self.imageCtrl = wx.StaticBitmap(self.panel, id=-1, bitmap=wxImg)
self.mainSizer = wx.BoxSizer(wx.VERTICAL)
self.mainSizer.Add(self.imageCtrl, 0, wx.ALL, 5)
self.panel.SetSizer(self.mainSizer)
self.mainSizer.Fit(self.frame)
self.panel.SetSizer(self.mainSizer)
self.panel.Layout()
self.frame.Show()
        self.MainLoop()
        return

#-----

if __name__ == '__main__':
    Test()

```

OTFfinder

```

# Support Module For DMA
#Dialog box for finding the Optical Transfer Function
#Shawn Gilliam, Teal Pershing 18 May 2012
#See TwoDFFTmodule.py
#http://wiki.wxpython.org/AnotherTutorial?highlight=%28commondialogs%29
# Hecht "Optics" textbook, transfer functions section...

```

```

import sys, time, struct
import inspect
import numpy.linalg as lin
import os.path
import ParameterPicker as pp
import wx
import os, sys
import numpy as np
import scipy as sp
import matplotlib.pyplot as plt
from PIL import Image

```

```

#-----for finding the Optical Transfer Function (OTF) using the Correlation Equation via Fourier Transform of an-----
#-----image divided by the Fourier Transform of the object. This gives the Fourier Transform of the Point -----
#-----Spread function which is the OTF.-----

```

```

#-----Note to the user-----
#-----I don't have time to make this friendly so it is your responsibility to make the FFT's that are being used-----
#-----have the same contour limits, sizes, etc, as chosen in the parameter picker section of the fourier transform-----
#-----program. I don't know what will happen if they are different but you are welcome to improve the program for -----
#-----your use.-----

```

```

class OTFdialog(wx.Dialog):
    def __init__(self, parent, id, title, creator):
        wx.Dialog.__init__(self, parent, id, title, wx.DefaultPosition, wx.Size(450, 280))

```

```

#-----
#-----

```

```

        dlg = wx.FileDialog(self, "Choose an image to use as Image FT", os.getcwd(), "", "", wx.OPEN)
        if dlg.ShowModal() == wx.ID_OK:
            mypath1 = dlg.GetPath()
            dlg.Destroy()

        self.corners= {'value': 1, 'units' : 'image (1 for yes)', 'description' : 'flip each quadrant about the y=x or y=-x line', 'minrange' : 0, 'maxrange' : 1}
        self.alimit= {'value': 500, 'units' : 'FFT', 'description' : 'Set maximum amplitude limit in FFT', 'minrange' : 0, 'maxrange' : 10000}
        self.contours= {'value': 10, 'units' : 'array', 'description' : 'Choose number of contour lines in graph', 'minrange' : 0, 'maxrange' : 100}
        self.parameters = {'Flip on Corners' : self.corners, 'Max amplitude' : self.alimit, 'contour lines' : self.contours}
        pd = pp.ParameterDialog(None, -1, 'Input Parameters for FFT', self) # + self.funk[f_choice].name + '
        pd.ShowModal()
        pd.Destroy()
        parameters=self.parameters
        print mypath1

        im1=Image.open(mypath1)
        xpix1=im1.size[0]          #xpix and ypix define the range of the input pattern array. im.size is tuple object, similar to lists in getting information from them
        #ypix=3*im.size[1]          #this is for larger filetypes; multiply the image size by the byte depth
        ypix1=im1.size[1]
        print xpix1

```

```

print ypix1
xcontour1=np.arange(0, xpix1, 1)
ycontour1=np.arange(0, ypix1, 1)
X1=range(xpix1)
Y1=range(ypix1)
imlist1=list(im1.getdata())
data1=np.array(imlist1)

pdashaped1=data1.reshape(ypix1, xpix1)
pdatarev1=pdashaped1[:,::-1]
parray1=pdatarev1
fftarray1=(np.log(np.abs(np.fft.fft2(parray1)))) #np.abs(np.fft.fft2(parray))

alimit=parameters['Max amplitude']['value'] #800 #sets an upper limit on the
possible values in the FFT array
cornerflip=parameters['Flip on Corners']['value'] #flips the values from the corners to the center, replicating an optical FT
print alimit
for y in Y1:
    for x in X1:
        #fftarray[y][x]=fftarray[y][x]/maxval
        if np.abs(fftarray1[y][x]) > alimit:
            fftarray1[y][x]=alimit

if cornerflip==1:
    print cornerflip
    a=fftarray1[0:int(ypix1/2), 0:int(xpix1/2)]
    c=fftarray1[ypix1/2:ypix1, 0:int(xpix1/2)]
    b=fftarray1[0:int(ypix1/2), (int(xpix1/2):xpix1)]
    d=fftarray1[(int(ypix1/2):ypix1), (int(xpix1/2):xpix1)]
    a=np.rot90(a, 2)
    b=np.rot90(b, 2)
    c=np.rot90(c, 2)
    d=np.rot90(d, 2)
    tophalf=np.hstack([a,b])
    bothalf=np.hstack([c,d])
    fftarray=np.vstack([tophalf, bothalf])
    print len(xcontour1)
    print len(ycontour1)

#print fftarray1

#-----

dlg = wx.FileDialog(self, "Choose an image to use as Object FT", os.getcwd(), "", "**.*", wx.OPEN)
if dlg.ShowModal() == wx.ID_OK:
    mypath2 = dlg.GetPath()
    dlg.Destroy()

self.pointsread= {'value': 0, 'units': 'Point Spread Function (1 for yes)', 'description': 'Plot PSF instead of OTF', 'minrange': 0, 'maxrange': 1}
self.corners= {'value': 1, 'units': 'image (1 for yes)', 'description': 'flip each quadrant about the y=x or y=-x line', 'minrange': 0, 'maxrange': 1}
self.alimit= {'value': 500, 'units': 'FFT', 'description': 'Set maximum amplitude limit in FFT', 'minrange': 0, 'maxrange': 10000}
self.contours= {'value': 10, 'units': 'array', 'description': 'Choose number of contour lines in graph', 'minrange': 0, 'maxrange': 100}
self.parameters = {'Flip on Corners': self.corners, 'Max amplitude': self.alimit, 'contour lines': self.contours, 'PSF': self.pointsread}
pd = pp.ParameterDialog(None, -1, 'Input Parameters for FFT', self) # + self.funk[f_choice].name + '
pd.ShowModal()
pd.Destroy()
parameters=self.parameters
print mypath2

im2=Image.open(mypath2)
xpix2=im2.size[0] #xpix and ypix define the range of the input pattern array. im.size is tuple object, similar to lists in getting information from them
#ypix=3*im.size[1] #this is for larger filetypes; multiply the image size by the byte depth
ypix2=im2.size[1]
print xpix2
print ypix2
xcontour2=np.arange(0, xpix2, 1)
ycontour2=np.arange(0, ypix2, 1)
X2=range(xpix2)
Y2=range(ypix2)
imlist2=list(im2.getdata())
data2=np.array(imlist2)

pdashaped2=data2.reshape(ypix2, xpix2)
pdatarev2=pdashaped2[:,::-1]
parray2=pdatarev2
fftarray2=(np.log(np.abs(np.fft.fft2(parray2)))) #np.abs(np.fft.fft2(parray))

alimit=parameters['Max amplitude']['value'] #800 #sets an upper limit on the
possible values in the FFT array
cornerflip=parameters['Flip on Corners']['value'] #flips the values from the corners to the center, replicating an optical FT
print alimit
for y in Y2:
    for x in X2:
        #fftarray[y][x]=fftarray[y][x]/maxval
        if np.abs(fftarray2[y][x]) > alimit:

```

```

                                fftarray2[y][x]=alimit
if cornerflip==1:
    print cornerflip
    a=fftarray2[0:int(ypix1/2) , 0:int(xpix1/2)]
    c=fftarray2[ypix1/2:ypix1 , 0:int(xpix1/2)]
    b=fftarray2[0:int(ypix1/2) , (int(xpix1/2)):xpix1]
    d=fftarray2[(int(ypix1/2)):ypix1 , (int(xpix1/2)):xpix1]
    a=np.rot90(a, 2)
    b=np.rot90(b, 2)
    c=np.rot90(c, 2)
    d=np.rot90(d, 2)
    tophalf=np.hstack([a,b])
    bothalf=np.hstack([c,d])
    fftarray=np.vstack([tophalf, bothalf])
    print len(xcontour2)
    print len(ycontour2)

# print fftarray2

#-----

OTF=[]
OTF=fftarray1/fftarray2
pointspread=parameters["PSF"]['value']          #flips the values from the corners to the center, replicating an optical FT
if pointspread==1:
    OTF=(np.log(np.abs(np.fft.ifft2(OTF))))          #this line performs the inverse fourier
transform of the OTF and should produce the point spread function

# but I am not convinced that it works. flipping the corners in is a closer approximation but more work is needed

# print '\n""OTF""\n'
# print OTF

clines=int(parameters["contour lines"]['value'])          #Determines the number of contour lines plotted
print clines
fplot=plt.contourf(xcontour1, ycontour1, OTF, clines)
plt.colorbar(fplot, shrink=0.8, extend='both')
plt.title("Optical Transfer Function of "+mypath1+"\n" and "+mypath2+"\n")          #Input Pattern'  #could use a way of replacing input
pattern with function's name
plt.show()

#-----

def WrongMod() :
    print 'This is not the module you want.  Execute control.py'

# Execute
if __name__ == '__main__':
    WrongMod()

```

TwoDFFTmodule

```

# Support Module For DMA
#Dialog box for loading bitmap images and displaying their FFT
#Shawn Gilliam and Teal Pershing 20 October 2011
#http://wiki.wxpython.org/AnotherTutorial?highlight=%28commondialogs%29

import sys, time, struct
import inspect
import numpy.linalg as lin
import os.path
import ParameterPicker as pp
import wx
import os, sys
import numpy as np
import scipy as sp
import matplotlib.pyplot as plt
from PIL import Image

#-----TwoDFFTmodule uses wx.Dialog to first load a bitmap image.  PIL is utilized to access the data from the -----
#-----image and find the image's size.  The numpy package function fft.ff2 then takes the two-dimensional FFT -----
#-----of the bitmap and plots it on a contour plot.  Refer to TwoDFourierTransformHann.py to see a stand-alone-----
#-----Two-dimensional FFT program and wx_model.py for using Windows Dialog Boxes. -----

class TwoDPowerSpectrum(wx.Dialog):
    def __init__(self, parent, id, title, creator):
        wx.Dialog.__init__(self, parent, id, title, wx.DefaultPosition, wx.Size(450, 280))
        dlg = wx.FileDialog(self, "Choose a bitmap to FFT", os.getcwd(), "", "", wx.OPEN)
        if dlg.ShowModal() == wx.ID_OK:

```

```

        mypath = dlg.GetPath()
        dlg.Destroy()
#dlg = wx.TextEntryDialog(self, 'Enter the maximum amplitude limit','Text Entry')
#dlg.SetValue("0")
#if dlg.ShowModal() == wx.ID_OK:
        #alimit=int(dlg.GetValue()) #self.SetStatusText("You entered: %s\n" % dlg.GetValue())
        #dlg.Destroy()
#self.dock = dock
#self.pp = self.dock.system_modules['ParameterPicker']
self.corners= {'value': 0, 'units': 'image (1 for yes)', 'description': 'flip each quadrant about the y=x or y=-x line', 'minrange': 0, 'maxrange': 1}
self.alimit= {'value': 500, 'units': 'FFT', 'description': 'Set maximum amplitude limit in FFT', 'minrange': 0, 'maxrange': 10000}
self.contours= {'value': 10, 'units': 'array', 'description': 'Choose number of contour lines in graph', 'minrange': 0, 'maxrange': 100}
self.parameters = {'Flip on Corners': self.corners, 'Max amplitude': self.alimit, 'contour lines': self.contours}
pd = pp.ParameterDialog(None, -1, 'Input Parameters for FFT', self) # + self.funk[f_choice].name + '
pd.ShowModal()
pd.Destroy()
parameters=self.parameters
print mypath
im=Image.open(mypath)
xpix=im.size[0]          #xpix and ypix define the range of the input pattern array. im.size is tuple object, similar to lists in getting information from them
ypix=3*im.size[1]          #this is for larger filetypes; multiply the image size by the byte depth
ypix=im.size[1]
print xpix
print ypix
xcontour=np.arange(0, xpix, 1)
ycontour=np.arange(0, ypix, 1)
X=range(xpix)
Y=range(ypix)
imlist=list(im.getdata())
#imlist.reverse()
#print imlist
#print len(imlist)
pdata=np.array(imlist)
pdatabased=pdata.reshape(ypix, xpix)
pdataberev=pdatabased[::-1,:].
parray=pdataberev
fftarray=(np.log(np.abs(np.fft.fft2(parray)))) #np.abs(np.fft.fft2(parray))
#maxval=fftarray.max()
alimit=parameters['Max amplitude']['value'] #800                                     #sets an upper limit on the
possible values in the FFT array
cornerflip=parameters['Flip on Corners']['value']          #flips the values from the corners to the center, replicating an optical FT
print alimit
for y in Y:
        for x in X:
                #fftarray[y][x]=fftarray[y][x]/maxval
                if np.abs(fftarray[y][x]) > alimit:
                        fftarray[y][x]=alimit
if cornerflip==1:
        print cornerflip
        a=fftarray[0:int(ypix/2) , 0:int(xpix/2)]
        c=fftarray[ypix/2:ypix , 0:int(xpix/2)]
        b=fftarray[0:int(ypix/2) , (int(xpix/2):xpix)]
        d=fftarray[(int(ypix/2):ypix) , (int(xpix/2):xpix)]
        a=np.rot90(a, 2)
        b=np.rot90(b, 2)
        c=np.rot90(c, 2)
        d=np.rot90(d, 2)
        tophalf=np.hstack([a,b])
        bothalf=np.hstack([c,d])
        fftarray=np.vstack([tophalf, bothalf])
        print len(xcontour)
        print len(ycontour)
clines=int(parameters['contour lines']['value'])          #Determines the number of contour lines plotted
print clines
fplot=plt.contour(xcontour, ycontour, fftarray, clines)
plt.colorbar(fplot, shrink=0.8, extend='both')
plt.title("Fast Fourier Transform of Input Pattern") #" +mypath+"\n"          Input Pattern) #could use a way of replacing input pattern with
function's name
plt.show()

class TwoDInvPowerSpectrum(wx.Dialog):
    def __init__(self, parent, id, title, creator):
        wx.Dialog.__init__(self, parent, id, title, wx.DefaultPosition, wx.Size(450, 280))
        dlg = wx.FileDialog(self, "Choose a bitmap to FFT", os.getcwd(), "", "", wx.OPEN)
        if dlg.ShowModal() == wx.ID_OK:
                mypath = dlg.GetPath()
                dlg.Destroy()
        print mypath
        im=Image.open(mypath)
        xpix=im.size[0]          #xpix and ypix define the range of the input pattern array. im.size is tuple object, similar to lists in getting information from them
        ypix=3*im.size[1]          #this is for larger filetypes; multiply the image size by the byte depth
        ypix=im.size[1]
        print xpix
        print ypix

```

```

xcontour=np.arange(0, xpix, 1)
ycontour=np.arange(0, ypix, 1)
X=range(xpix)
Y=range(ypix)
imlist=list(im.getdata())
pdata=np.array(imlist)
pdatabshaped=pdata.reshape(ypix, xpix)
pdatarev=pdatabshaped[ ::-1, :]
parray=pdatarev
iftfarray=np.log(np.abs(np.fft.ifft2(parray)))
alimit=200 #sets an upper limit on the possible values in the FFT array
for y in Y:
    for x in X:
        if np.abs(iftfarray[y][x]) > alimit:
            ftfarray[y][x]=alimit
            #if ftfarray[y][x]=1
        #if else:
            #if ftfarray[y][x]=0
#a=iftfarray[0:int(ypix/2) , 0:int(xpix/2)]
#c=iftfarray[ypix/2:ypix , 0:int(xpix/2)]
#b=iftfarray[0:int(ypix/2) , (int(xpix/2)):xpix]
#d=iftfarray[(int(ypix/2)):ypix , (int(xpix/2)):xpix]
#a=np.rot90(a, 2)
#b=np.rot90(b, 2)
#c=np.rot90(c, 2)
#d=np.rot90(d, 2)
#tophalf=np.hstack([a,b])
#bothalf=np.hstack([c,d])
#ftfarray=np.vstack([tophalf, bothalf])
clines=50 #Determines the number of contour lines plotted
fplot=plt.contour(xcontour, ycontour, iftfarray, clines)
plt.colorbar(fplot, shrink=0.8, extend='both')
plt.title('Inverse FFT of Input Pattern') #could use a way of replacing input pattern with function's name
plt.show()

```

```

def WrongMod() :
    print 'This is not the module you want. Execute control.py'

```

```

# Execute
if __name__ == '__main__':
    #TwoDPowerSpectrum()
    WrongMod()

```

bitmap_functions

```

compatability = ['bitmap_control v 0.3', 'bitmap_control v 1.0']

```

```

import sys, time, struct
import inspect
import numpy as np
import numpy.linalg as lin
import wx
import os.path

```

```

def ListFunctions(module) :
    # See http://code.activestate.com/recipes/553262-list-classes-methods-and-functions-in-a-module/ for more extensive examples.
    fl = []
    for name in dir(module):
        obj = getattr(module, name)
        if inspect.isclass(obj):
            #print obj.__name__
            fl.append(obj())
    return fl

```

```

class PeriodicEllipsesSolid :
    def __init__(self) :
        self.name = 'Periodic Ellipses Solid'
        self.description = 'cos(kx)*exp(-k*abs(y-y0))'
        #-----default parameters-----
        self.k = {'value' : 2, 'units' : '1/meters', 'description' : 'wave number', 'minrange' : 0, 'maxrange' : 10}
        self.y = {'value' : 383, 'units' : 'pixels', 'description' : 'y-coordinate', 'minrange' : 0, 'maxrange' : 767}
        self.ftdo = {'value': 0, 'units' : 'image', 'description' : 'fourier-transform execute', 'minrange' : 0, 'maxrange' : 1}
        self.alimit= {'value': 0, 'units' : 'FFT', 'description' : 'Set maximum amplitude limit in FFT', 'minrange' : 0, 'maxrange' : 10000 }
        self.parameters = {'Wave Number' : self.k, 'yo' : self.y, 'FFT' : self.ftdo, 'Max amplitude' : self.alimit}
        self.error = "
    def func(self, X, Y, parameters) :
        error = "
        elapsed_time = 0
        #-----works fine without this for and if else loop.....what is it for?-----
        for q in self.parameters :

```

```

        if q in parameters :
            self.parameters[q] = parameters[q]
        else :
            error = 'Error. ' + q + ' not found in parameter dictionary.'
            print error
            return error, np.zeros(len(X) * len(Y)), 0

farray = np.array([])
k = parameters['Wave Number'] * np.pi / 200.0
yo = parameters['yo'] * [value]
start_time = time.time()
for y in Y :
    factor = np.exp(-k * np.abs(y - yo))
    farray = np.append(farray, np.cos(k * X) * factor)
elapsed_time = time.time() - start_time
return error, farray, elapsed_time

class CircularSinc :
    def __init__(self) :
        self.name = 'Circular Sinc'
        self.description = 'sinc(kr), r=abs(x-xo, y-yo)'
        self.k = {'value' : 1, 'units' : '1/meters', 'description' : 'wave number', 'minrange' : 0, 'maxrange' : 10}
        self.x = {'value' : 511, 'units' : 'pixels', 'description' : 'x-coordinate', 'minrange' : 0, 'maxrange' : 1023}
        self.y = {'value' : 383, 'units' : 'pixels', 'description' : 'y-coordinate', 'minrange' : 0, 'maxrange' : 767}
        self.fftdo = {'value' : 0, 'units' : 'image', 'description' : 'fourier-transform execute', 'minrange' : 0, 'maxrange' : 1}
        self.alimit = {'value' : 0, 'units' : 'FFT', 'description' : 'Set maximum amplitude limit in FFT', 'minrange' : 0, 'maxrange' : 10000}
        self.parameters = {'Wave Number' : self.k, 'xo' : self.x, 'yo' : self.y, 'FFT' : self.fftdo, 'Max amplitude' : self.alimit}
        self.error = ""

    def func(self, X, Y, parameters) :
        error = ""
        elapsed_time = 0
        for q in self.parameters :
            if q in parameters :
                self.parameters[q] = parameters[q]
            else :
                error = 'Error. ' + q + ' not found in parameter dictionary.'
                print error
                return error, np.zeros(len(X) * len(Y)), 0

        farray = np.zeros(len(X) * len(Y))
        k = parameters['Wave Number'] * np.pi / 200.0
        xo = parameters['xo'] * [value]
        yo = parameters['yo'] * [value]
        start_time = time.time()
        X2 = np.power(X - xo, 2)
        Y2 = np.power(Y - yo, 2)
        i = 0
        for y in Y2 :
            for x in X2 :
                r = np.sqrt(x + y)
                farray[i] = np.sinc(k * r)      #sin(k*r)/k/r
                i += 1

        elapsed_time = time.time() - start_time
        return error, farray, elapsed_time

class SingleCircle :
    def __init__(self) :
        self.name = 'Single Blocking Circle'
        self.description = 'R, r=abs(x-xo, y-yo)'
        self.x = {'value' : 511, 'units' : 'pixels', 'description' : 'x-coordinate', 'minrange' : 0, 'maxrange' : 1023}
        self.y = {'value' : 383, 'units' : 'pixels', 'description' : 'y-coordinate', 'minrange' : 0, 'maxrange' : 767}
        self.radius = {'value' : 150, 'units' : 'pixels', 'description' : 'radius of circle', 'minrange' : 0, 'maxrange' : 1000}
        self.fftdo = {'value' : 0, 'units' : 'image', 'description' : 'fourier-transform execute', 'minrange' : 0, 'maxrange' : 1}
        self.alimit = {'value' : 0, 'units' : 'FFT', 'description' : 'Set maximum amplitude limit in FFT', 'minrange' : 0, 'maxrange' : 10000}
        self.parameters = {'xo' : self.x, 'yo' : self.y, 'R' : self.radius, 'FFT' : self.fftdo, 'Max amplitude' : self.alimit}
        self.error = ""

    def func(self, X, Y, parameters) :
        error = ""
        elapsed_time = 0
        for q in self.parameters :
            if q in parameters :
                self.parameters[q] = parameters[q]
            else :
                error = 'Error. ' + q + ' not found in parameter dictionary.'
                print error
                return error, np.zeros(len(X) * len(Y)), 0

        farray = np.zeros(len(X) * len(Y))
        xo = parameters['xo'] * [value]
        yo = parameters['yo'] * [value]
        R = parameters['R'] * [value]
        start_time = time.time()
        X2 = np.power(X - xo, 2)
        Y2 = np.power(Y - yo, 2)
        i = 0
        for y in Y2 :
            for x in X2 :
                r = np.sqrt(x + y)

```



```

        if r>R:
            farray[i] = 1 #sin(k*r)/k/r
        i += 1
    elapsed_time = time.time() - start_time
    return error, farray, elapsed_time

class CircularOmission :
    def __init__(self) :
        self.name = 'Two Circular Masks'
        self.description = 'R, r=abs(x-xo, y-yo)'
        self.x1 = {'value': 333, 'units': 'pixels', 'description': 'x-coordinate', 'minrange': 0, 'maxrange': 1023}
        self.y1 = {'value': 383, 'units': 'pixels', 'description': 'y-coordinate', 'minrange': 0, 'maxrange': 767}
        self.radius1 = {'value': 77, 'units': 'pixels', 'description': 'radius of circle', 'minrange': 0, 'maxrange': 1000}
        self.x2 = {'value': 666, 'units': 'pixels', 'description': 'x-coordinate', 'minrange': 0, 'maxrange': 1023}
        self.y2 = {'value': 383, 'units': 'pixels', 'description': 'y-coordinate', 'minrange': 0, 'maxrange': 767}
        self.radius2 = {'value': 88, 'units': 'pixels', 'description': 'radius of circle', 'minrange': 0, 'maxrange': 1000}
        self.fftdo = {'value': 0, 'units': 'image', 'description': 'fourier-transform execute', 'minrange': 0, 'maxrange': 1}
        self.alimit = {'value': 0, 'units': 'FFT', 'description': 'Set maximum amplitude limit in FFT', 'minrange': 0, 'maxrange': 10000}
        self.parameters = {'xo1': self.x1, 'yo1': self.y1, 'R1': self.radius1, 'xo2': self.x2, 'yo2': self.y2, 'R2': self.radius2, 'FFT': self.fftdo, 'Max amplitude' :

self.alimit}

        self.error = ""
    def func(self, X, Y, parameters) :
        error = ""
        elapsed_time = 0
        for q in self.parameters :
            if q in parameters :
                self.parameters[q] = parameters[q]
            else :
                error = 'Error. ' + q + ' not found in parameter dictionary.'
                print error
                return error, np.zeros(len(X) * len(Y)), 0

        farray = np.ones(len(X) * len(Y))
        xo1 = parameters['xo1']['value']
        yo1 = parameters['yo1']['value']
        R1 = parameters['R1']['value']
        xo2 = parameters['xo2']['value']
        yo2 = parameters['yo2']['value']
        R2 = parameters['R2']['value']
        start_time = time.time()
        X1 = np.power(X - xo1, 2)
        Y1 = np.power(Y - yo1, 2)
        X2 = np.power(X - xo2, 2)
        Y2 = np.power(Y - yo2, 2)
        i = 0
        for y in Y1 :
            for x in X1 :
                r = np.sqrt(x + y)
                if r<R1 :
                    farray[i] = 0 #sin(k*r)/k/r
                i += 1
        i = 0
        for y in Y2 :
            for x in X2 :
                r = np.sqrt(x + y)
                if r<R2 :
                    farray[i] = 0 #sin(k*r)/k/r
                i += 1
        elapsed_time = time.time() - start_time
        return error, farray, elapsed_time

class SingleBlackCircle :
    def __init__(self) :
        self.name = 'Single Transmitting Circle'
        self.description = 'R, r=abs(x-xo, y-yo)'
        self.x = {'value': 511, 'units': 'pixels', 'description': 'x-coordinate', 'minrange': 0, 'maxrange': 1023}
        self.y = {'value': 383, 'units': 'pixels', 'description': 'y-coordinate', 'minrange': 0, 'maxrange': 767}
        self.radius = {'value': 150, 'units': 'pixels', 'description': 'radius of circle', 'minrange': 0, 'maxrange': 1000}
        self.fftdo = {'value': 0, 'units': 'image', 'description': 'fourier-transform execute', 'minrange': 0, 'maxrange': 1}
        self.alimit = {'value': 0, 'units': 'FFT', 'description': 'Set maximum amplitude limit in FFT', 'minrange': 0, 'maxrange': 10000}
        self.parameters = {'xo': self.x, 'yo': self.y, 'R': self.radius, 'FFT': self.fftdo, 'Max amplitude' : self.alimit}
        self.error = ""
    def func(self, X, Y, parameters) :
        error = ""
        elapsed_time = 0
        for q in self.parameters :
            if q in parameters :
                self.parameters[q] = parameters[q]
            else :
                error = 'Error. ' + q + ' not found in parameter dictionary.'
                print error
                return error, np.zeros(len(X) * len(Y)), 0

        farray = np.zeros(len(X) * len(Y))
        xo = parameters['xo']['value']
        yo = parameters['yo']['value']

```

```

R = parameters['R']['value']
start_time = time.time()
X2 = np.power(X - xo, 2)
Y2 = np.power(Y - yo, 2)
i = 0
for y in Y2 :
    for x in X2 :
        r = np.sqrt(x + y)
        if r<R:
            farray[i] = 1 #sin(k*r)/k/r
        i += 1
elapsed_time = time.time() - start_time
return error, farray, elapsed_time

```

class FilterAnnulus :

```

def __init__(self) :
    self.name = 'Blocking Annulus'
    self.description = 'R, r=abs(x-xo, y-yo)'
    self.x = {'value': 511, 'units': 'pixels', 'description': 'x-coordinate', 'minrange': 0, 'maxrange': 1023}
    self.y = {'value': 383, 'units': 'pixels', 'description': 'y-coordinate', 'minrange': 0, 'maxrange': 767}
    self.radius = {'value': 100, 'units': 'pixels', 'description': 'radius of inner circle', 'minrange': 0, 'maxrange': 1000}
    self.bigradius = {'value': 150, 'units': 'pixels', 'description': 'radius of second circle', 'minrange': 0, 'maxrange': 1000}
    self.fftdo = {'value': 0, 'units': 'image', 'description': 'fourier-transform execute', 'minrange': 0, 'maxrange': 1}
    self.alimit = {'value': 0, 'units': 'FFT', 'description': 'Set maximum amplitude limit in FFT', 'minrange': 0, 'maxrange': 10000}
    self.parameters = {'xo': self.x, 'yo': self.y, 'rr': self.radius, 'R': self.bigradius, 'FFT': self.fftdo, 'Max amplitude': self.alimit}
    self.error = ""

def func(self, X, Y, parameters) :
    error = ""
    elapsed_time = 0
    for q in self.parameters :
        if q in parameters :
            self.parameters[q] = parameters[q]
        else :
            error = 'Error. ' + q + ' not found in parameter dictionary.'
            print error
            return error, np.zeros(len(X) * len(Y)), 0

    farray = np.zeros(len(X) * len(Y))
    xo = parameters['xo']['value']
    yo = parameters['yo']['value']
    rr = parameters['rr']['value']
    R = parameters['R']['value']
    start_time = time.time()
    X2 = np.power(X - xo, 2)
    Y2 = np.power(Y - yo, 2)
    i = 0
    for y in Y2 :
        for x in X2 :
            r = np.sqrt(x + y)
            if r>R or r<rr:
                farray[i] = 1 #sin(k*r)/k/r
            i += 1

    #for y in Y2 :
        #for x in X2 :
            #r = np.sqrt(x + y)
            #if r<rr:
                #farray[i] = 0 #sin(k*r)/k/r
            #i += 1

    elapsed_time = time.time() - start_time
    return error, farray, elapsed_time

```

class TransmittingAnulus :

```

def __init__(self) :
    self.name = 'Transmitting Anulus'
    self.description = 'R, r=abs(x-xo, y-yo)'
    self.x = {'value': 511, 'units': 'pixels', 'description': 'x-coordinate', 'minrange': 0, 'maxrange': 1023}
    self.y = {'value': 383, 'units': 'pixels', 'description': 'y-coordinate', 'minrange': 0, 'maxrange': 767}
    self.radius = {'value': 100, 'units': 'pixels', 'description': 'radius of inner circle', 'minrange': 0, 'maxrange': 1000}
    self.bigradius = {'value': 150, 'units': 'pixels', 'description': 'radius of second circle', 'minrange': 0, 'maxrange': 1000}
    self.fftdo = {'value': 0, 'units': 'image', 'description': 'fourier-transform execute', 'minrange': 0, 'maxrange': 1}
    self.alimit = {'value': 0, 'units': 'FFT', 'description': 'Set maximum amplitude limit in FFT', 'minrange': 0, 'maxrange': 10000}
    self.parameters = {'xo': self.x, 'yo': self.y, 'rr': self.radius, 'R': self.bigradius, 'FFT': self.fftdo, 'Max amplitude': self.alimit}
    self.error = ""

def func(self, X, Y, parameters) :
    error = ""
    elapsed_time = 0
    for q in self.parameters :
        if q in parameters :
            self.parameters[q] = parameters[q]
        else :
            error = 'Error. ' + q + ' not found in parameter dictionary.'
            print error
            return error, np.zeros(len(X) * len(Y)), 0

    farray = np.zeros(len(X) * len(Y))
    xo = parameters['xo']['value']
    yo = parameters['yo']['value']

```

```

rr = parameters['rr']['value']
R = parameters['R']['value']
start_time = time.time()
X2 = np.power(X - xo, 2)
Y2 = np.power(Y - yo, 2)
i = 0
for y in Y2 :
    for x in X2 :
        r = np.sqrt(x + y)
        if rr<r<R:
            farray[i] = 1 #sin(k*r)/k/r
        i += 1
    #for y in Y2 :
        #for x in X2 :
            #r = np.sqrt(x + y)
            #if r<rr:
                #farray[i] = 0 #sin(k*r)/k/r
            #i += 1
elapsed_time = time.time() - start_time
return error, farray, elapsed_time

class ConcentricTransmittingAnnuli :
    def __init__(self) :
        self.name = 'Concentric Annuli'
        self.description = 'R, r=abs(x-xo, y-yo)'
        self.x = {'value': 511, 'units': 'pixels', 'description': 'x-coordinate', 'minrange': 0, 'maxrange': 1023}
        self.y = {'value': 383, 'units': 'pixels', 'description': 'y-coordinate', 'minrange': 0, 'maxrange': 767}
        self.radius1 = {'value': 100, 'units': 'pixels', 'description': 'radius of inner circle', 'minrange': 0, 'maxrange': 1000}
        self.bigradius1 = {'value': 150, 'units': 'pixels', 'description': 'radius of second circle', 'minrange': 0, 'maxrange': 1000}
        self.radius2 = {'value': 200, 'units': 'pixels', 'description': 'radius of inner circle', 'minrange': 0, 'maxrange': 1000}
        self.bigradius2 = {'value': 250, 'units': 'pixels', 'description': 'radius of second circle', 'minrange': 0, 'maxrange': 1000}
        self.radius3 = {'value': 300, 'units': 'pixels', 'description': 'radius of inner circle', 'minrange': 0, 'maxrange': 1000}
        self.bigradius3 = {'value': 350, 'units': 'pixels', 'description': 'radius of second circle', 'minrange': 0, 'maxrange': 1000}
        self.ftdo = {'value': 0, 'units': 'image', 'description': 'fourier-transform execute', 'minrange': 0, 'maxrange': 1}
        self.alimit = {'value': 0, 'units': 'FFT', 'description': 'Set maximum amplitude limit in FFT', 'minrange': 0, 'maxrange': 10000}
        self.parameters = {'xo': self.x, 'yo': self.y, 'rr1': self.radius1, 'R1': self.bigradius1, 'rr2': self.radius2, 'R2': self.bigradius2, 'rr3': self.radius3, 'R3':
self.bigradius3, 'FFT': self.ftdo, 'Max amplitude': self.alimit}
        self.error = ""
    def func(self, X, Y, parameters) :
        error = ""
        elapsed_time = 0
        for q in self.parameters :
            if q in parameters :
                self.parameters[q] = parameters[q]
            else :
                error = 'Error. ' + q + ' not found in parameter dictionary.'
                print error
                return error, np.zeros(len(X) * len(Y)), 0
        farray = np.zeros(len(X) * len(Y))
        xo = parameters['xo']['value']
        yo = parameters['yo']['value']
        rr1 = parameters['rr1']['value']
        R1 = parameters['R1']['value']
        rr2 = parameters['rr2']['value']
        R2 = parameters['R2']['value']
        rr3 = parameters['rr3']['value']
        R3 = parameters['R3']['value']
        start_time = time.time()
        X2 = np.power(X - xo, 2)
        Y2 = np.power(Y - yo, 2)
        i = 0
        for y in Y2 :
            for x in X2 :
                r = np.sqrt(x + y)
                if rr1<r<R1 or rr2<r<R2 or rr3<r<R3:
                    farray[i] = 1 #sin(k*r)/k/r
                i += 1
            #for y in Y2 :
                #for x in X2 :
                    #r = np.sqrt(x + y)
                    #if r<rr:
                        #farray[i] = 0 #sin(k*r)/k/r
                    #i += 1
        elapsed_time = time.time() - start_time
        return error, farray, elapsed_time

class RectangularOmission :
    def __init__(self) :
        self.name = 'Rectangular omitting blocks'
        self.description = 'R, r=abs(x-xo, y-yo)'
        self.x1block1 = {'value': 0, 'units': 'pixels', 'description': 'left side of first rectangle', 'minrange': 0, 'maxrange': 1024}
        self.x2block1 = {'value': 1024, 'units': 'pixels', 'description': 'right side of first rectangle', 'minrange': 0, 'maxrange': 1024}
        self.y1block1 = {'value': 0, 'units': 'pixels', 'description': 'top of first rectangle', 'minrange': 0, 'maxrange': 768}
        self.y2block1 = {'value': 768, 'units': 'pixels', 'description': 'bottom of first rectangle', 'minrange': 0, 'maxrange': 768}
        self.x1block2 = {'value': 0, 'units': 'pixels', 'description': 'left side of second rectangle', 'minrange': 0, 'maxrange': 1024}

```

```

self.x2block2 = {'value': 1024, 'units': 'pixels', 'description': 'right side of second rectangle', 'minrange': 0, 'maxrange': 1024}
self.y1block2 = {'value': 0, 'units': 'pixels', 'description': 'top of second rectangle', 'minrange': 0, 'maxrange': 768}
self.y2block2 = {'value': 768, 'units': 'pixels', 'description': 'bottom of second rectangle', 'minrange': 0, 'maxrange': 768}
self.fftdo= {'value': 0, 'units': 'image', 'description': 'fourier-transform execute', 'minrange': 0, 'maxrange': 1}
self.alimit= {'value': 0, 'units': 'FFT', 'description': 'Set maximum amplitude limit in FFT', 'minrange': 0, 'maxrange': 10000}
self.parameters = {'x1b1': self.x1block1, 'x2b1': self.x2block1, 'y1b1': self.y1block1, 'y2b1': self.y2block1, 'x1b2': self.x1block2, 'x2b2': self.x2block2,
'y1b2': self.y1block2, 'y2b2': self.y2block2, 'FFT': self.fftdo, 'Max amplitude': self.alimit}
self.error = ""
def func(self, X, Y, parameters):
    error = ""
    elapsed_time = 0
    for q in self.parameters:
        if q in parameters:
            self.parameters[q] = parameters[q]
        else:
            error = 'Error. ' + q + ' not found in parameter dictionary.'
            print error
            return error, np.ones(len(X) * len(Y)), 0

    farray = np.ones(len(X) * len(Y))
    x1b1 = parameters['x1b1']['value']
    x2b1 = parameters['x2b1']['value']
    y1b1 = parameters['y1b1']['value']
    y2b1 = parameters['y2b1']['value']
    x1b2 = parameters['x1b2']['value']
    x2b2 = parameters['x2b2']['value']
    y1b2 = parameters['y1b2']['value']
    y2b2 = parameters['y2b2']['value']
    start_time = time.time()
    #X2 = np.power(X - xo, 2)
    #Y2 = np.power(Y - yo, 2)
    i=0
    for y in Y:
        for x in X:
            if y1b1 < y < y2b1 and x1b1 < x < x2b1:
                farray[i] = 0 #sin(k*r)/k/r
            if y1b2 < y < y2b2 and x1b2 < x < x2b2:
                farray[i] = 0 #sin(k*r)/k/r
            i += 1

    elapsed_time = time.time() - start_time
    return error, farray, elapsed_time

class RectangularTransmission:
    def __init__(self):
        self.name = 'Rectangular transmitting blocks'
        self.description = 'R, r=abs(x-xo, y-yo)'
        self.x1block1 = {'value': 0, 'units': 'pixels', 'description': 'left side of first rectangle', 'minrange': 0, 'maxrange': 1024}
        self.x2block1 = {'value': 1024, 'units': 'pixels', 'description': 'right side of first rectangle', 'minrange': 0, 'maxrange': 1024}
        self.y1block1 = {'value': 0, 'units': 'pixels', 'description': 'top of first rectangle', 'minrange': 0, 'maxrange': 768}
        self.y2block1 = {'value': 768, 'units': 'pixels', 'description': 'bottom of first rectangle', 'minrange': 0, 'maxrange': 768}
        self.x1block2 = {'value': 0, 'units': 'pixels', 'description': 'left side of second rectangle', 'minrange': 0, 'maxrange': 1024}
        self.x2block2 = {'value': 1024, 'units': 'pixels', 'description': 'right side of second rectangle', 'minrange': 0, 'maxrange': 1024}
        self.y1block2 = {'value': 0, 'units': 'pixels', 'description': 'top of second rectangle', 'minrange': 0, 'maxrange': 768}
        self.y2block2 = {'value': 768, 'units': 'pixels', 'description': 'bottom of second rectangle', 'minrange': 0, 'maxrange': 768}
        self.fftdo= {'value': 0, 'units': 'image', 'description': 'fourier-transform execute', 'minrange': 0, 'maxrange': 1}
        self.alimit= {'value': 0, 'units': 'FFT', 'description': 'Set maximum amplitude limit in FFT', 'minrange': 0, 'maxrange': 10000}
        self.parameters = {'x1b1': self.x1block1, 'x2b1': self.x2block1, 'y1b1': self.y1block1, 'y2b1': self.y2block1, 'x1b2': self.x1block2, 'x2b2': self.x2block2,
'y1b2': self.y1block2, 'y2b2': self.y2block2, 'FFT': self.fftdo, 'Max amplitude': self.alimit}
        self.error = ""
    def func(self, X, Y, parameters):
        error = ""
        elapsed_time = 0
        for q in self.parameters:
            if q in parameters:
                self.parameters[q] = parameters[q]
            else:
                error = 'Error. ' + q + ' not found in parameter dictionary.'
                print error
                return error, np.zeros(len(X) * len(Y)), 0

        farray = np.zeros(len(X) * len(Y))
        x1b1 = parameters['x1b1']['value']
        x2b1 = parameters['x2b1']['value']
        y1b1 = parameters['y1b1']['value']
        y2b1 = parameters['y2b1']['value']
        x1b2 = parameters['x1b2']['value']
        x2b2 = parameters['x2b2']['value']
        y1b2 = parameters['y1b2']['value']
        y2b2 = parameters['y2b2']['value']
        start_time = time.time()
        #X2 = np.power(X - xo, 2)
        #Y2 = np.power(Y - yo, 2)
        i=0
        for y in Y:
            for x in X:
                if y1b1 < y < y2b1 and x1b1 < x < x2b1:

```



```

        elapsed_time = time.time() - start_time
        return error, farray, elapsed_time

class Fascinating :
    def __init__(self):
        self.name = 'Fascinating'
        self.description = 'Rn^2 = nF*lambda, An = n*pi&F*lambda, F = focal length, all lengths in meters.'
        self.f = {'value': 1.0, 'units': 'meters', 'description': 'focal length', 'minrange': 0, 'maxrange': 10}
        self.l = {'value': 633, 'units': 'nanometers', 'description': 'wavelength', 'minrange': 450, 'maxrange': 850}
        self.fftdo = {'value': 0, 'units': 'image', 'description': 'fourier-transform execute', 'minrange': 0, 'maxrange': 1}
        self.alimit = {'value': 0, 'units': 'FFT', 'description': 'Set maximum amplitude limit in FFT', 'minrange': 0, 'maxrange': 10000}
        self.parameters = {'Focal Length': self.f, 'Wavelength': self.l, 'FFT': self.fftdo, 'Max amplitude': self.alimit}
        self.error = ""

    def func(self, X, Y, parameters):
        error = ""
        elapsed_time = 0
        for q in self.parameters:
            if q in parameters:
                self.parameters[q] = parameters[q]
            else:
                error = 'Error. ' + q + ' not found in parameter dictionary.'
                print error
                return error, np.zeros(len(X) * len(Y)), 0

        farray = np.zeros(len(X) * len(Y))
        wl = parameters['Wavelength']['value']/1000000000
        f = parameters['Focal Length']['value']
        fac = f*wl
        start_time = time.time()
        i = 0
        for y in Y:
            z = np.power(y, 2)
            for x in X:
                r2 = np.power(x, 2) + z
                if int(r2/fac)%2 == 0:
                    farray[i] = 1
                i += 1
            elapsed_time = time.time() - start_time
        return error, farray, elapsed_time

class FresnelZonePlateCylindrical :
    def __init__(self):
        self.name = 'Fresnel Zone Plate Cylindrical'
        self.description = 'Rn^2 = nF*lambda, An = n*pi&F*lambda, F = focal length, all lengths in meters.'
        self.f = {'value': 1.0, 'units': 'meters', 'description': 'focal length', 'minrange': 0, 'maxrange': 10}
        self.l = {'value': 633, 'units': 'nanometers', 'description': 'wavelength', 'minrange': 450, 'maxrange': 850}
        self.x = {'value': 511, 'units': 'pixels', 'description': 'x-coordinate', 'minrange': 0, 'maxrange': 1023}
        self.fftdo = {'value': 0, 'units': 'image', 'description': 'fourier-transform execute', 'minrange': 0, 'maxrange': 1}
        self.alimit = {'value': 0, 'units': 'FFT', 'description': 'Set maximum amplitude limit in FFT', 'minrange': 0, 'maxrange': 10000}
        self.parameters = {'Focal Length': self.f, 'Wavelength': self.l, 'xo': self.x, 'FFT': self.fftdo, 'Max amplitude': self.alimit}
        self.error = ""

    def func(self, X, Y, parameters):
        error = ""
        elapsed_time = 0
        for q in self.parameters:
            if q in parameters:
                self.parameters[q] = parameters[q]
            else:
                error = 'Error. ' + q + ' not found in parameter dictionary.'
                print error
                return error, np.zeros(len(X) * len(Y)), 0

        farray = np.zeros(len(X) * len(Y))
        wl = parameters['Wavelength']['value']/1000000000
        f = parameters['Focal Length']['value']
        xo = parameters['xo']['value']
        fac = f*wl
        start_time = time.time()
        X2 = np.power((X - xo) * 1.3e-05, 2)      # Each pixel is 13 um
        i = 0
        for y in Y:
            for x in X2:
                if int(x/fac)%2 == 0:
                    farray[i] = 1
                i += 1
            elapsed_time = time.time() - start_time
        return error, farray, elapsed_time

class UserDiffraction :
    def __init__(self):
        self.name = 'User Defined Diffraction Grating'
        self.description = ""
        self.f = {'value': 1.0, 'units': 'meters', 'description': 'focal length', 'minrange': 0, 'maxrange': 10}
        self.l = {'value': 600, 'units': 'nanometers', 'description': 'wavelength', 'minrange': 450, 'maxrange': 850}
        self.k = {'value': 1, 'units': '1/meters', 'description': 'wave number', 'minrange': 0, 'maxrange': 10}
        self.x = {'value': 511, 'units': 'pixels', 'description': 'x-coordinate', 'minrange': 0, 'maxrange': 1023}
        self.y = {'value': 383, 'units': 'pixels', 'description': 'y-coordinate', 'minrange': 0, 'maxrange': 767}

```

```

self.fftdo= {'value': 0, 'units': 'image', 'description': 'fourier-transform execute', 'minrange': 0, 'maxrange': 1}
self.alimit= {'value': 0, 'units': 'FFT', 'description': 'Set maximum amplitude limit in FFT', 'minrange': 0, 'maxrange': 10000}
self.parameters = {'Focal Length': self.f, 'Wavelength': self.l, 'Wave Number': self.k, 'xo': self.x, 'yo': self.y, 'FFT': self.fftdo, 'Max amplitude': self.alimit}
self.error = ""
def func(self, X, Y, parameters):
    error = ""
    elapsed_time = 0
    for q in self.parameters:
        if q in parameters:
            self.parameters[q] = parameters[q]
        else:
            error = 'Error. ' + q + ' not found in parameter dictionary.'
            print error
            return error, np.zeros(len(X) * len(Y)), 0
    farray = np.zeros(len(X) * len(Y))
    wl = parameters['Wavelength']['value']/1000000000
    f = parameters['Focal Length']['value']
    xo = parameters['xo']['value']
    yo = parameters['yo']['value']
    k = parameters['Wave Number']['value']*np.pi/200.0
    fac = f*wl
    start_time = time.time()
    X2 = np.power((X - xo) * 1.3e-05, 2)      # Each pixel is 13 um
    i = 0
    for y in Y:
        for x in X2:
            if int(x/fac)%2 == 0:
                farray[i] = 1
                i += 1
            #----- k does not seem to effect the outcome, play with this later-----
            factor = np.exp(-k*np.abs(y-yo))
            farray = np.append(farray, np.cos(k*X) * factor)
    elapsed_time = time.time() - start_time
    return error, farray, elapsed_time

```

#-----The following section is an attempt to make it easier for a new user to come in and add a new function to-----
#-----our library and to make it fun at the same time, the fun part is still under construction so hopefully its easy----

```

#class MyFunction:
#    def __init__(self):
#        self.name = 'Name of MyFunction'
#        self.description = ''
#
#        ##-----These are dictionaries that hold the values and information that go along with that value-----
#        ##-----the value key in your dictionary is the value that will be entered in the parameter input box-----
#        ##-----the units are the descriptive of what the value should be, description is what the value talks about-----
#        ##-----and the min and max range are the range of values that will be allowed in your function-----
#        self.f = {'value': defaultvalues, 'units': 'meters', 'description': 'focal length', 'minrange': 0, 'maxrange': 10}
#        self.l = {'value': defaultvalues, 'units': 'meters', 'description': 'wavelength', 'minrange': 450, 'maxrange': 850}
#        self.k = {'value': defaultvalues, 'units': '1/meters', 'description': 'wave number', 'minrange': 0, 'maxrange': 10}
#        self.x = {'value': defaultvalues, 'units': 'pixels', 'description': 'x-coordinate', 'minrange': 0, 'maxrange': 1023}
#        self.y = {'value': defaultvalues, 'units': 'pixels', 'description': 'y-coordinate', 'minrange': 0, 'maxrange': 767}
#        self.fftdo = {'value': 0, 'units': 'image', 'description': 'fourier-transform execute', 'minrange': 0, 'maxrange': 1}
#
#        ##-----This is a dictionary of the previous dictionaries, you should have a dictionary for each parameter -----
#        ##-----that you want to be able to change in your function-----
#        self.parameters = {'Focal Length': self.f, 'Wavelength': self.l, 'Wave Number': self.k, 'xo': self.x, 'yo': self.y, 'FFT': self.fftdo}
#        self.error = ""
#
#    def func(self, X, Y, parameters):
#        ##-----This section is an error checker for the parameters of the function skip down-----
#        error = ""
#        elapsed_time = 0
#        for q in self.parameters:
#            if q in parameters:
#                self.parameters[q] = parameters[q]
#            else:
#                error = 'Error. ' + q + ' not found in parameter dictionary.'
#                print error
#                return error, np.zeros(len(X) * len(Y)), 0
#        farray = np.zeros(len(X) * len(Y))
#
#        ##-----The following is where you set the values from your function to the values of the dictionary-----
#        ##-----wl is wavelength in nanometers so the value entered in the parameter input box is divided by 1000000000-----
#        wl = parameters['Wavelength']['value']/1000000000
#        f = parameters['Focal Length']['value']
#        xo = parameters['xo']['value']
#        yo = parameters['yo']['value']
#        k = parameters['Wave Number']['value']*np.pi/200.0
#
#        ##-----This section deals with the math of your function and you must work that out on your own, good luck!-----
#        fac = f*wl

```

```

#start_time = time.time()
#X2 = np.power((X - xo) * 1.3e-05, 2)    # Each pixel is 13 um
#i = 0
#for y in Y :
#    #for x in X2 :
#        #if int(x/fac)%2 == 0 :
#            #farray[i] = 1
#            #i += 1
#        ##----- k does not seem to effect the outcome, play with this later-----
#        #factor = np.exp(-k*np.abs(y-yo))
#        #farray = np.append(farray, np.cos(k*X) * factor)

##-----Other housekeeping and a return statement, shouldnt concern you-----
#elapsed_time = time.time() - start_time
#return error, farray, elapsed_time

```

```

#-----
def WrongMod() :
    print 'This is not the module you want.  Execute control.py'
#-----

# Execute
if __name__ == '__main__':
    WrongMod()

```


Appendix B – Fourier Transform Laboratory Exercise

PH 481

Physical Optics

Winter 2012

Laboratory #9

Week of March 12

Read: pp. 519-529, 539-544 of "Optics" by Hecht
Do: 1. Experiment IX.1: : Fourier Transform of a Ronchi Ruling

Experiment IX.1: Fourier Transform of a Ronchi Ruling

The goal of this lab is to analyze the Fraunhofer diffraction pattern of an image and compare to the two-dimensional Fourier transform of the same image. *The field distribution in the Fraunhofer diffraction pattern is the Fourier transform of the field distribution across the aperture.** Figure 1 shows a diagram of the apparatus that will be used in the laboratory exercise.

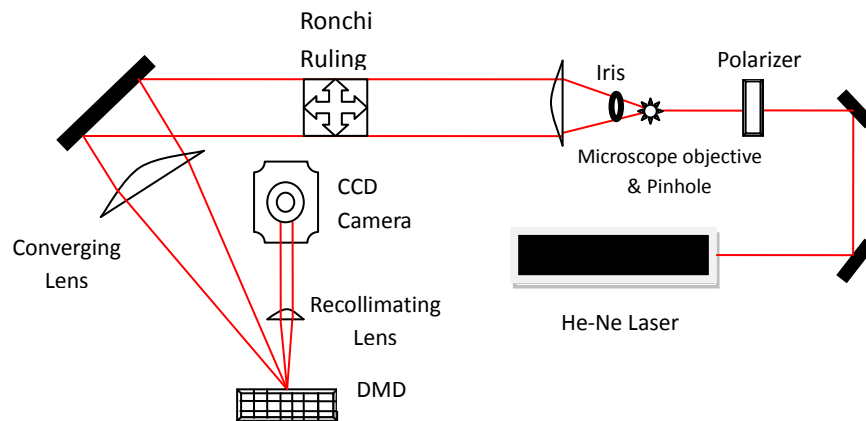


Figure 1: Diagram of optical Fourier analysis setup.

The apparatus consists of a series of lenses and mirrors that allow for experimental Fourier analysis using a Digital Micro-mirror Array (DMA). The laser light is collimated using a beam expanding telescope consisting of a microscope objective, iris, and converging lens. A spatial filter (in this experiment, a Ronchi ruling) is placed in the object plane to produce an optical pattern. A converging lens then places the Fraunhofer diffraction pattern, which is also the optical Fourier transform, onto the focal plane. *Keeping in mind that each point on the focal plane corresponds to a spatial frequency in the object plane*, the DMA is used as a spatial filter in the focal plane to analyze the intensity of any particular spatial frequency. The selected spatial frequency is recollimated using another converging lens and viewed using a CCD camera.

For this experiment, a Ronchi ruling will be used as the object plane spatial filter. Recall that the position for each spatial frequency component k_x and k_y in the Fourier transform plane is found with

$$k_x = \frac{kX}{R} \quad \text{and} \quad k_y = \frac{kY}{R} ,$$

where k is the optical wave number and R is the focal length of the lens (for this experiment, $R = 500 \text{ mm}$). From the pre-lab homework problem, you have already calculated the distance for each intensity maxima in the viewing plane; use the separation of each mirror on the DMA ($10.8 \mu\text{m} = 1 \text{ pixel}$) to find the distance of each intensity maxima from the DMA's center in pixels.

Now that we have found the location of each intensity maxima on the DMA, we seek to measure the intensity of each maxima experimentally. The correct spatial filter must be selected to transmit the desired spatial frequency component. In the D4100 GUI, Click the "Load" tab on the left and press "open image." A library of usable spatial filters is now available to you; you may choose to use either an annuli, two dots, or two horizontal bars as your filter shape. Which shape did you choose and why?

First, select the "all-black" image. This will transmit all light incident on the DMA to the CCD camera. In the "load" tab, choose "load and reset" and "global" as the loading arguments and click "Add." Click the "control" tab and add a delay time command of $\sim 5000 \text{ ms}$. Click the "run" button in the top control bar to run the script. Open the IC Capture software and take a picture. Save the image; this will be used as a reference to the individual spatial frequency intensities.

We will now experimentally measure the intensity of each frequency component produced by the Ronchi ruling using the DMA. Using your calculated spatial frequency locations, repeat the above process with a new image that will transmit only the zero harmonic in the focal plane (that is, the central bright spot of the Fourier transform). How does the image produced by the zeroth order harmonic filter differ from that produced by the all-transmitting filter?

Repeat the process for next three neighboring spatial frequency components in the Fourier transform plane. Be sure to save images for each spatial frequency!

We will now open the images for each spatial frequency and analyze the maximum intensity using ImageJ. First, open the image of the zeroth order harmonic and use the "analyze" feature to determine the maximum intensity measured by the CCD. Repeat this operation with each image collected for the spatial frequencies. Plot the intensity as a function of spatial frequency (k_x). How do the relative values compare with the max amplitudes predicted in the theoretical model?

NOTE: Our diagram should state explicitly where the Focal plane and Object plane are in our experimental setup.

Equipment needed:

Item	Qty	Source (part #)
Helium-Neon Laser	1	Melles Griot 05 LHP 121
Al mirror	3	Newport 10D10ER.1
Polarizer	1	Edmund A38,396
200 mm lens	2	Newport KPX106
500 mm lens	1	Newport KPX118
150 mm lens	1	Newport KPX100
Magnetic Lens Post	1	Newport MB-2
Mounting posts	8	Thor Labs P3
Translation Stage	2	Newport 423
Iris	1	Newport
Ronchi Ruling	1	
Digital Micromirror D.	1	Texas Instruments
CCD Camera	1	Imaging Center DMK21AU04
Power supply	1	Uniphase 1205

Pre-Lab Homework Problem

A planar electric field filtered with a Ronchi ruling is functionally represented in the object plane as a series of rectangle functions of the form

$$f(x) = E_0 e^{-i\omega t} \sum_{n=0}^N \text{rect}\left(\frac{x - (-aN + 2an)}{a}\right)$$

where a is the width of the rectangle, N is the number of transmitting bars in the ruling, and $x_0 = -aN + 2an$ is the center of each bar as shown in Figure 1.

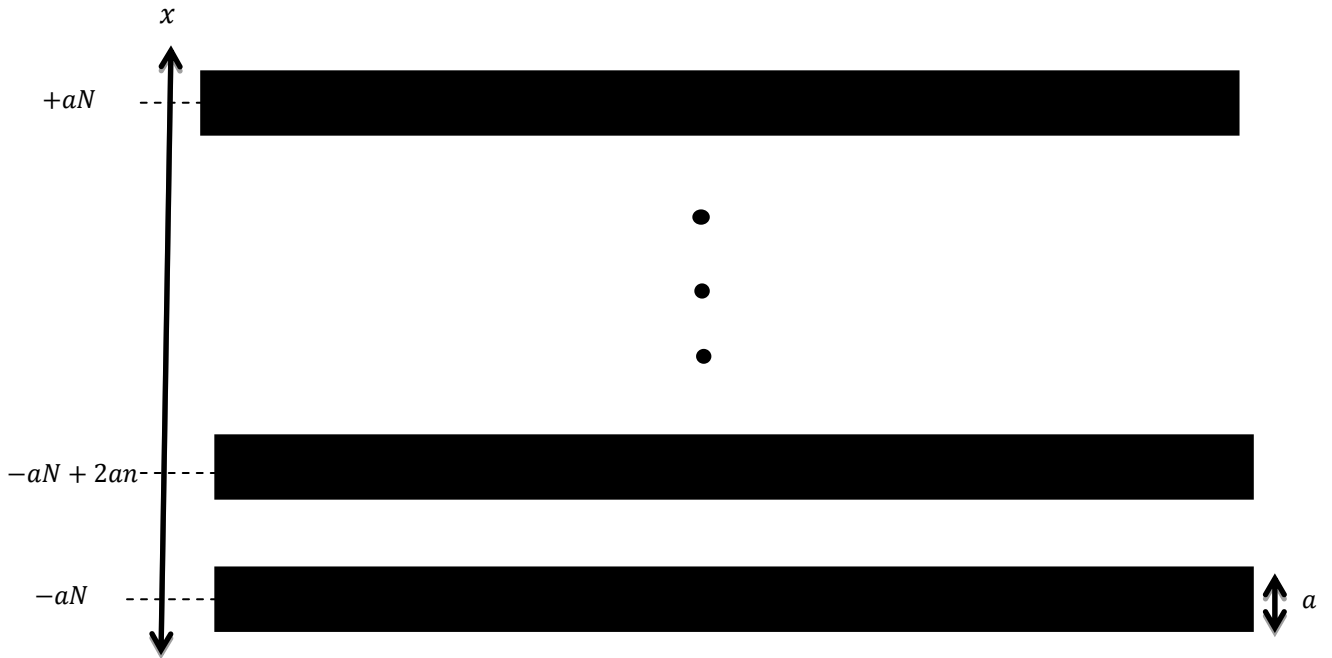


Figure 1: Diagram of a Ronchi ruling.

- Compute the spatial Fourier transform of the electric field in the object plane.
- Find the intensity $I(k_x)$ of the electric field resulting from the Fourier transform. Compare this solution to the intensity pattern produced by multiple slits in the Fraunhofer regime after making the change of variables

$$k_x = \frac{kX}{R} ,$$

where X is the position in the viewing plane and R is the distance between the object plane and the point X in the viewing plane (Note: recall that $X/R = \sin \theta$).

- Sketch the intensity pattern for $N=4$ as a function of X/R .
- Using your answer from part b) find the position of the first four maxima when

$$\begin{aligned} a &= 0.0208 \text{ cm} , \\ R &= 50 \text{ cm} , \\ \lambda &= 633 \text{ nm} . \end{aligned}$$

- Find the ratio between the central maximum intensity and each of the next three maximum intensities.

Experiment Data:

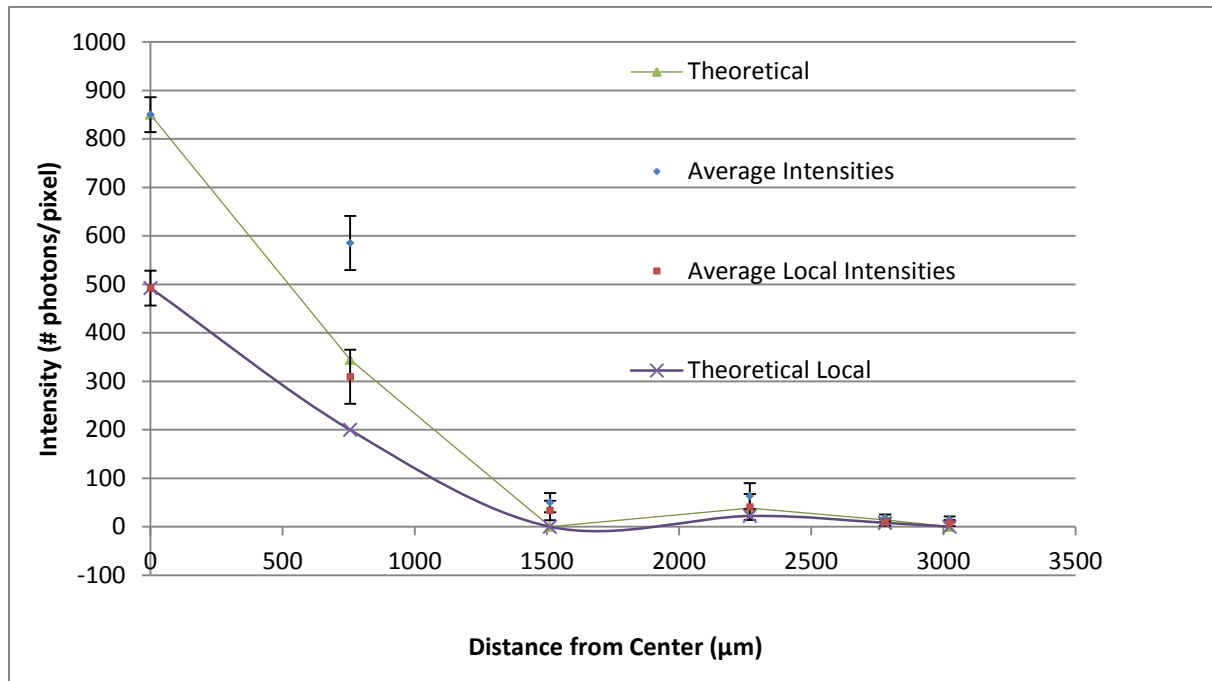


FIGURE 4: THIS IS AN AVERAGE OF THE DATA THAT WAS FOUND EXPERIMENTALLY. THE DIFFERENCE IN THEORETICAL DATA AND EXPERIMENTAL SHOULD BE EXPLAINED USING APODIZATION.

Appendix B – Fresnel Zone Plates Laboratory Exercise

The Fresnel Zone Plate experiment was conducted to obtain knowledge of the capabilities of the DMA to perform in this task. It is left to the reader to form conclusions to the usefulness of the instruments. All images are labeled and is the extent of the information that was recorded.

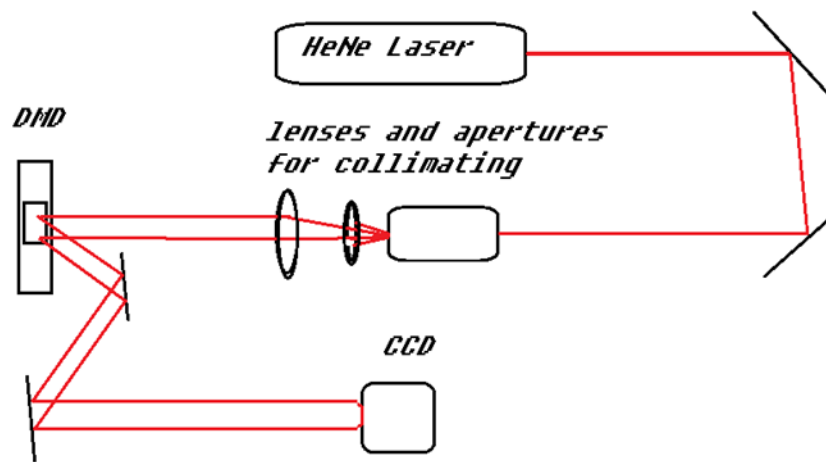


FIGURE 1: DIAGRAM OF SETUP

Fresnel Zone Plate Masks

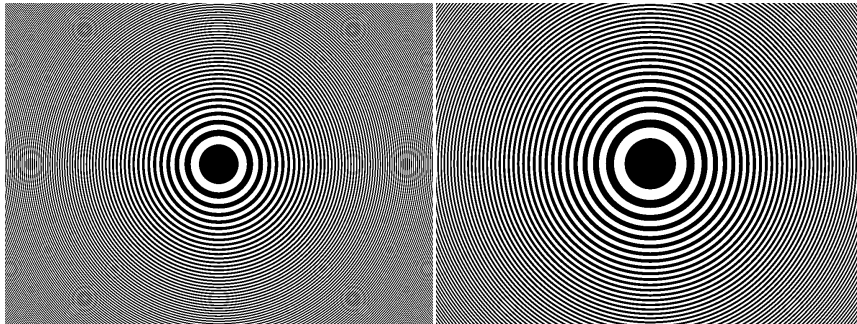


FIGURE 2: PATTERN FOR 0.6 M FOCAL LENGTH FIGURE 3: PATTERN FOR 1 M FOCAL LENGTH

0.6 Meter Focal Length Data

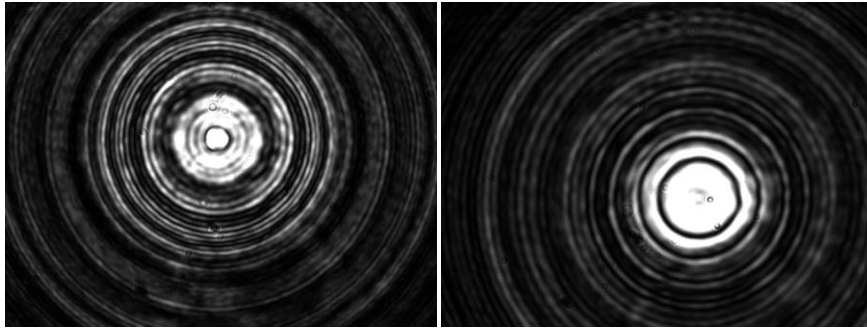


FIGURE 4: 0.6 M FOCAL LENGTH AT 0.2 M FIGURE 5: 0.6 M FOCAL LENGTH AT 0.3 M

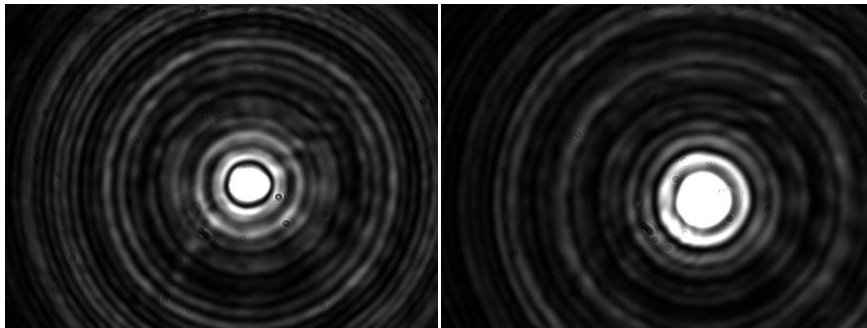


FIGURE 6: 0.6 M FOCAL LENGTH AT 0.4 M FIGURE 7: 0.6 M FOCAL LENGTH AT 0.5 M

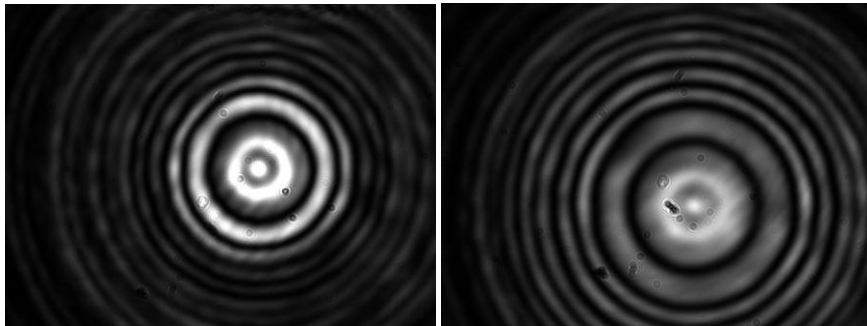


FIGURE 8: 0.6 M FOCAL LENGTH AT 0.6 M FIGURE 9: 0.6 M FOCAL LENGTH AT 0.7 M

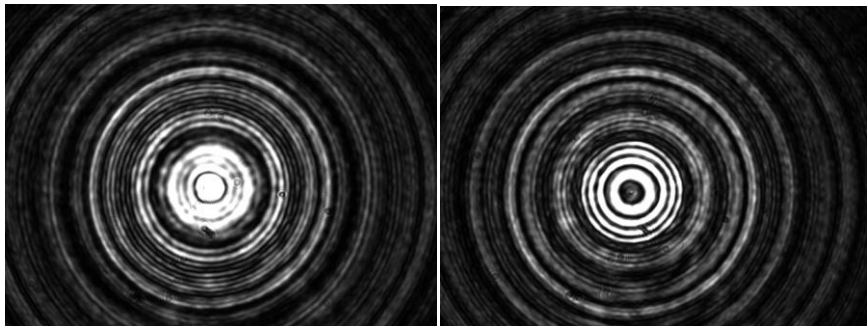


FIGURE 10: 0.6 M FOCAL LENGTH AT 0.19 M FIGURE 11: 0.6 M FOCAL LENGTH AT 0.18 M

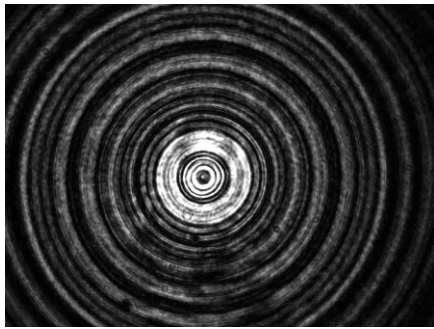


FIGURE 12: 0.6 M FOCAL LENGTH AT 0.08 M

1 Meter Focal Length Data

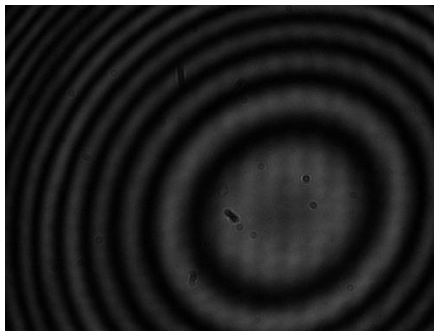


FIGURE 13: 1 M FOCAL LENGTH AT 1.2 M

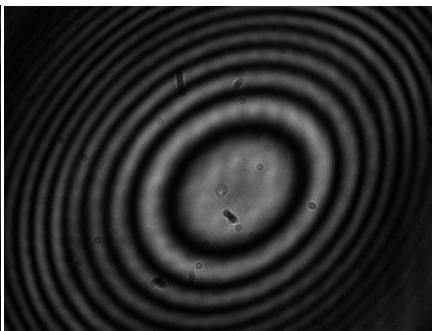


FIGURE 14: 1 M FOCAL LENGTH AT 1.0 M

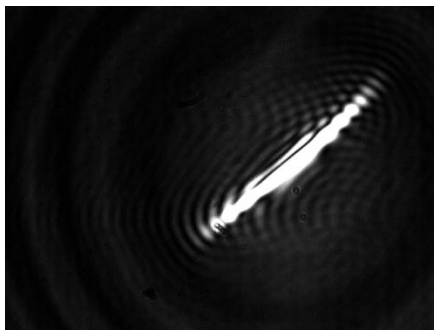


FIGURE 15: 1 M FOCAL LENGTH AT 0.8 M

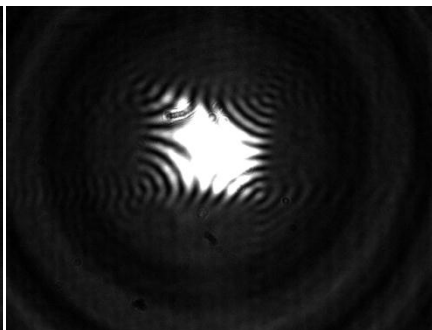


FIGURE 16: 1 M FOCAL LENGTH AT 0.7 M

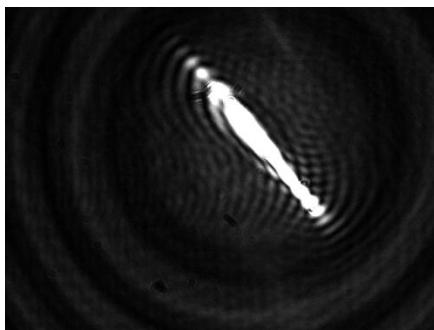


FIGURE 17: 1 M FOCAL LENGTH AT 0.65 M

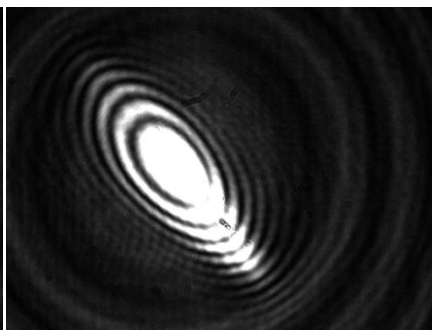


FIGURE 18: 1 M FOCAL LENGTH AT 0.6 M

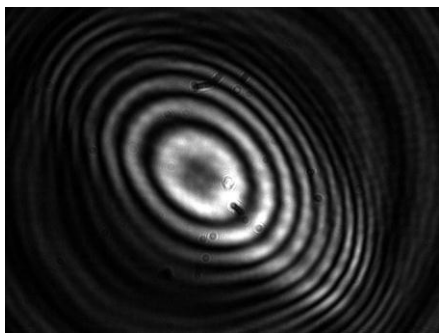


FIGURE 19: 1 M FOCAL LENGTH AT 0.5 M

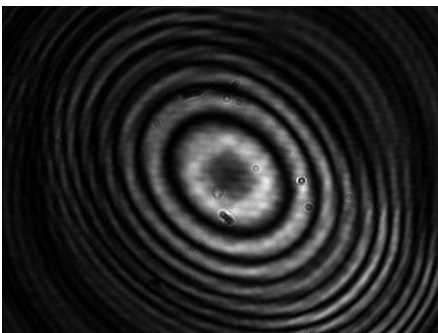


FIGURE 50: 1 M FOCAL LENGTH AT 0.45 M

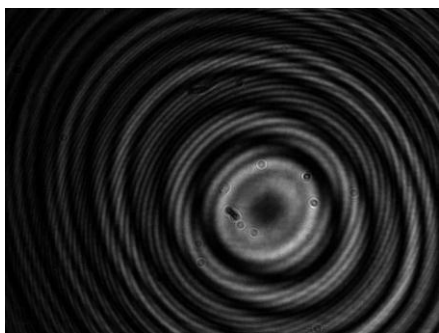


FIGURE 21: 1 M FOCAL LENGTH AT 0.3 M

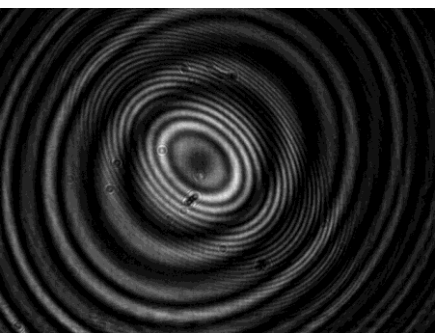


FIGURE 22: 1 M FOCAL LENGTH AT 0.25 M

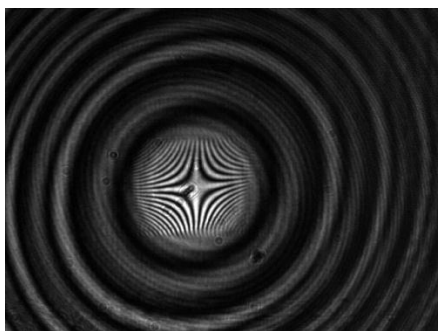


FIGURE 23: 1 M FOCAL LENGTH AT 0.2 M

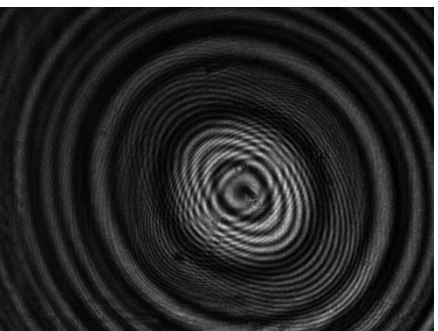


FIGURE 24: 1 M FOCAL LENGTH AT 0.15 M

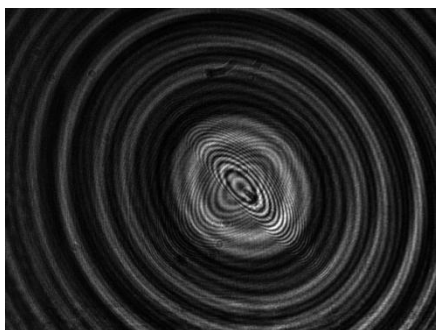


FIGURE 25: 1 M FOCAL LENGTH AT 0.1 M

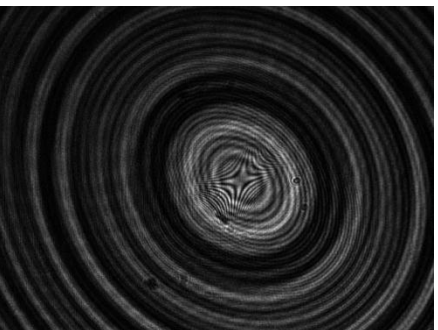


FIGURE 26: 1 M FOCAL LENGTH AT 0.08 M