1 Introduction

My project goal was to simulate light diffraction through a small arbitrarily shaped aperture and displaying the resulting diffraction pattern. I chose this topic because I had just covered waves and optics in Physics 5B and was interested in the more complex physics behind it. I had multiple simulations, each building on each other. However, in the end I was able to simulate light diffraction through an aperture defined by a grey scale image while making it interactive, allowing control over multiple parameters, and also simulate white light (a combination of wavelengths) diffraction. I will use 2 pages as just background of my models, so feel free to read Thought Process and Methods section for my actual report and reference Models if need be. Sorry this is very long, but I wanted to cover my entire from start to end!

2 Thought Process and Methods

When starting this project, I had no idea where to start, so I started with the simplest cases which had many approximations. While this limited the ability of my simulator, it allowed me to get an idea of how to make more advanced simulations.

I used Fraunhofer Diffraction first, which as detailed 4.1 is simply sinc^2 where in Python all the variables are parameters of my function. This model allowed me an easy way to use methods in Python I already knew, specifically Numpy arrays, Matplotlib, and for loops. My idea was to construct a meshgrid with a specified resolution (size of array) and defined dimensions of width and height to represent my screen. Then, construct a ones Numpy array with again resolution and dimension to represent my aperture. I then figured I can use two nested for loops to iterate over the points in the aperture array, calculating the individual sinc^2 functions for the x and y axes on the screen. My function would just take in the x and y of my screen meshgrid This then involved multiplying them together and adding to an empty Numpy array with the same resolution as the screen. This works because each component of the meshgrid is just an array of x or y coordinates of the same shape. I then plotted my linspaces with the resultant array with my diffraction pattern using pcolormesh, allowing me to have the correct dimensions of my axes. My vmax then served as a "sensitivty" which I can tune to view more of the pattern.

When doing this originally, I found it difficult at the time to get a good diffraction pattern, which I found was due to my use of a Numpy ones array for my aperture. It was actually better to use a linspace for the x and y axes of the aperture, allowing me to iterate over the y then the x to account for every point. Note, for all of these linspaces and meshgrids, they have origins centered in the middle which is a constant throughout all my later simulations,

which for this simulation required using floor division and some "size factors" manipulating my specified dimensions and resolutions.

My next simulation used Fresnel Diffraction, detailed in 4.2. This was more involved since it required Fourier Transforms. Luckily, Scipy has these functions built in as Fast Fourier Transforms, as well as a multitude of other functions related to the FFT making some processes later more efficient and easier.

Approaching this, I wanted to have more flexibility in the shape of my aperture. I had the idea of allowing images to be imported into my script, then being resized and made into a Numpy array to be used as my $E(x', y', 0)$ function. These images would be forced into grey scale and reformatted so that black was 0 and anything else (greater than 0) was 1 (swapping 0 and 1 and using a circle can show the Arago spot[1]!). I did this by using the Pillow[2] package, which allows for image processing. To note, I used a universal resolution of my simulation that applies to both the aperture and screen. My process was to import an image as grayscale, create a black image with the same resolution, find a "size factor" from $\frac{\text{size}_{\text{aperture}}}{\text{size}_{\text{screen}}}$ and multiply to resolution and use to resize aperture, paste aperture onto black image, and convert into Numpy array. This was very difficult to figure out since I had to learn a new package and think of how to reshape my aperture appropriately. What helped me was reading the docs and really understand the limitations of each function I needed.

I then found I can use Scipy's .fftshift to find my spatial frequencies ω Since I want these to have dimension and a spatial frequency for each point on the screen, I use my resolution as the number of spatial frequencies and space them using $\frac{\text{dimension}}{\text{resolution}}$. I can then put these into a meshgrid like the screen in the last simulation when ω_x and ω_y are found. Since $k_z = 2\pi\sqrt{(\frac{1}{\lambda})^2 - w_x^2 - w_y^2}$, I simply use the meshgrid x and y components and plug them into k_z. I then used the fft2 and fftshift functions find my final diffraction pattern. Note that in the models section, I have specific formulas for ω, however, fftfreq works just fine. Finally, I used imshow and a formula for the extent which I found on an online resource, however I can't find it again. The vmax I made to be just the maximum value of the diffraction pattern, so no data was lost. However this was multiplied by a "sensitivity" variable so I could see more of the pattern if I wanted (less than one was brighter).

I use the same variables as Fraunhofer Diffraction which are all input into the Fresnel Integral. I also ensure to check the Fresnel condition mentioned in the Wikipedia.

My last and final simulation used the Angular Spectrum Method, detailed in 4.3. I wanted to make this as accurate and interactive as possible. I had two different components; a simulator that allowed for adjustment of different parameters, while only displaying a single

[1] Arago Spot
[2] Pillow Package Documentation

wavelength (true to color if wanted!) and a simulator that propagates white light.

I first worked on getting the core component, the Fourier Transforms, working. I took what I had learned from the Fresnel Diffraction simulation (fftfreq, fft2, ifft2, and fftshift), and applied many of the same methods for this. I also used the same aperture function to take in an image for the aperture. To find k_z I again used fftfreq, but then plugged them into $k_z = 2\pi\sqrt{(\frac{1}{\lambda})^2 - (\omega_x)^2 - (\omega_y)^2}$. However, to account for the square root potentially being imaginary I used the Numpy .where function to make up for this, taking the absolute square root instead and multiplying the resulting k_z by $1j$ any time the argument under the square root is negative (same size arrays so .where still works). This seems like the fastest method, since .where is very efficient (rather than more packages just to get an imaginary square root function). Finally, I used the Numpy .real and .conjugate to find the final intensity array and divide it by the max value to normalize to 1. Luckily, my screen size is the same as my aperture plane dimensions so my extent is computed like my aperture aperture linspaces in my Fraunhofer simulator.

To display this diffraction pattern with interactivity, I actually decided to turn to online resources. I found a Stack Exchange[3] forum with a solution that uses classes. However, this required me to make my program object oriented. So, what I did was make a class for my diffraction and then a separate class for the plotter. I reorganized my code into attributes, method for the propagation, method for the aperture, then methods that let me get certain attributes. The plotter from the forum also required a method that returns a list of tuples for properties of the slider, where each tuple is organized into name of variable, range for the variable, and then I added an element for the name that I want for the slider. I require that when making the Diffraction object, only the resolution and units used for the aperture and screen plane are permanently set. Everything else has initial values to be changed by the sliders. A trouble I ran into was my sliders having scientific notation when displaying the current variable value, since I was directly adjusting variables that had units on them. I separated setting my value and applying my units, which required putting my wavelength, aperture dimension, and screen dimension unit application into my propagation instead so that it displayed the raw value without units instead.

What my plotter then does is make subplots in it's attributes, then load my Diffraction object and use my propagation method. The initial plot is made with imshow, and my axis labels change depending on what unit the user wants to use for the aperture and screen planes. I then added a title that displays my wavelength using python string formatting. Another method creates the sliders using values from the method from Diffraction that returns slider properties. It shifts up the plot and spaces out the slider, appending each slider to a list that will be plotted by the plotter. An update function then uses setattr to load the variable

[3] *matplotlib - Add sliders to a figure dynamically*

changes and use set_data on the imshow plot to change the diffraction array. I added also conidtions that checks what the name of the variable being added is, and if it changes extent or wavelength, it uses set_extent and set_title to change it dynamically. Finally it uses .draw to refresh the plot. This is then used in the plotter in a for loop, iterating through each tuple in the slider property list to make each slider and uses on_change to run the update function and apply it to the plot.

I then wanted to make my diffraction pattern to change wavelength. To do this, I used some colorimetery and color science, which luckily has many of the required functions in the colour[4] package! I use CIE colormatching functions[5] and sRGB conversions to get my colors so that they display on a screen as our eyes would see them. The process from wavelength to sRGB color is

1. Wavelength to CIE XYZ functions
2. XYZ functions to sRGB linear
3. sRGB linear to sRGB (requiring gamma corrections and truncating to (0,1) range as sRGB requires)

This is made into 3 separate functions which are run in a 4th function to simply be used in my plotter to run my propagation and the current wavelength through it. I then added a boolean parameter to my plotter to check if I want color or just the pattern and adjust accordingly using if statements, instead of having two plot methods.

The last part of my simulator was my white light propagation. I made a separate class with it's own plotter for this simulation. This was simple enough, I just for loop through my propogation through a specified wavelength range a specified amount of times (which was made into parameters in my plotter), applying my color function, and adding to a zeros array. However, this would give me a really bright diffraction pattern. I then found out about CIE illuminants, which specified the brightness of each wavelength as perceived by our eyes. I essentially take the CIE illuminant values (for D65[6] specifically), normalize to 1 for the largest value, and divide by the number of iterations of my "plot stacking" function I talked about in the beginning of the paragraph. The appropriate value for the current wavelength was then simply multiplied onto my wavelength to XYZ function output.

My plotter for this class was actually simpler than the single color simulation, taking everything but the sliders. Luckily, imshow takes RGB data in a NxMx3 array, so this plotting was easy!

[4] Colour Package
[5] Wikipedia page on CIE colorspace
[6] D65 Illuminant

3 Conclusion

What I learned was that creating a simulator requires a lot of just implementing theory into Python and dealing with discrete values rather than continuous. I also learned a lot about OOP and have a new appreciation for it (a lot more organized!). This also exposed me to many new packages, and taught me how to read documentation and learn them on my own. I think if I were to extend this project, I would work more on optimization to make the plotter update much quicker while homogenizing my classes to reuse my propagation for both single color and white diffraction using inheritance. I would also make my white propagation more accurate color wise, since diffraction should look more spread out in terms of colors (more rainbow-y).

4 Models

4.1 Fraunhofer Diffraction

My first simulator used "Fraunhofer Diffraction"[7] which is an approximation of the Fresnel Diffraction Integral.

What Fraunhofer Diffraction requires is that, for an aperture of width a, a screen distance from the aperture of length L, and a light wave of wavelength λ

$$\frac{a^2}{\lambda} << L \text{ and } a << L. \tag{1}$$

which set my limiting conditions for my parameters.

I used the Fraunhofer Diffraction approximation, which for the intensity of the diffraction pattern at any point (x,y) on the screen using a square turns out to be (after some integrals and such)

$$I(x,y) \propto \text{sinc}^2(\frac{\pi W x}{\lambda R})\text{sinc}^2(\frac{\pi H y}{\lambda R})^{8} \tag{2}$$

where R is the distance from the center of the aperture to the point on the screen, (W, H) is the width and height of the aperture, and sinc is the function $\frac{\sin(x)}{x}$.

4.2 Fresnel Diffraction

My next simulator used the Fresnel Diffraction Integral[9], which is a solution to the famous Helmholtz equation ($\nabla^2\Psi = -k^2\Psi$ where k is the *wave number*). We use approximations to

[7] Wikipedia page on Fraunhofer Diffraction
[8] Wikipedia page on Fraunhofer Diffraction Equation for Rectangular Aperture
[9] Wikipedia page on Fresnel Diffraction

arrive at conditions

$$\lambda << z \text{ and } \lambda << \rho \tag{3}$$

where ρ is the difference between the radius in the screen plane and the aperture plane respectively. Essentially, the dimensions of the setup must be much bigger than the wavelength. It can be simplified to be proportional to a Fourier Transform (using a bunch of substitution, expansion, and pulling of factors out of the integral)

$$E(x,y,z) \propto \mathscr{F}[E(x',y',0)e^{i\frac{\pi}{\lambda z}(x'^2+y'^2)}](\omega_x = \frac{x}{\lambda z}, \omega_y = \frac{y}{\lambda z}) \tag{4}$$

where E is the electric field, (x,y) is the coordinates of the screen plane, z is the distance between the aperture and screen and ω is just the spatial frequencies required in a Fourier Transform.

4.3 Angular Spectrum Method

The angular spectrum method[10] uses the Fourier Transform too. The basic idea to this method is:

1. Construct an electric field acting as our light wave (emanating from the aperture plane)
2. Take the 2D FT of the electric field
3. Multiply this by a propagation term to account for the phase change light undergoes as it travels
4. Inverse 2D FT to get back a potentially complex electric field
5. Multiply by the conjugate to get back intensity

This mathematically looks like

$$U(x,y,z) = \mathscr{F}[E(x',y',0)](\omega_x,\omega_y) \tag{5}$$

$$I'(x,y,z) = \mathscr{F}^{-1}[U(x,y,z)e^{ik_z z}] \tag{6}$$

$$I(x,y,z) = \bar{I'}I' \tag{7}$$

where k_z is the z component of the wave vector which is $k_z = 2\pi\sqrt{(\frac{1}{\lambda})^2 - \omega_x^2 - \omega_y^2}$ for this method. Again, ω are spatial frequencies, similar to Fresnel spatial frequencies but defined differently. What this method allows is virtually no limiting conditions as in Fraunhofer or Fresnel Diffraction, meaning we can take any distance and any wavelength we want!

[10] Libretexts Article on Angular Spectrum Method