Fresnel Diffraction from an Aperture

Sarah Nicole Straw

11th November 2024 — Word count: 1503

Abstract

This report investigates the application of two numerical integration techniques to model diffraction patterns in the Fresnel approximation. Diffraction of light around an aperture is analyzed by calculating the electric field and therefore intensity of light at varying distances and screen coordinates. Two numerical integration approaches are employed: Gaussian quadrature, implemented with SciPy's dblquad function, and the Monte Carlo integration method coded independently. Parameters aperture size and screen distance are varied to explore the transition between Fresnel (near-field) and Fraunhofer (far-field) diffraction. Results demonstrate the advantages and limitations of each integration method, with Gaussian quadrature providing control over integration accuracy and Monte Carlo techniques offering flexibility for irregular aperture shapes. The resulting diffraction patterns reveal insights into the interplay between screen distance, aperture size, and diffraction intensity, highlighting the capabilities of numerical methods for solving complex integrals in physical optics.

1 Introduction

Diffraction is a fundamental phenomenon in wave optics that occurs when light encounters an obstacle or aperture, leading to interference patterns that depend on the geometry of the aperture and the distance to the observation screen. The diffraction pattern can be described using the Fresnel and Fraunhofer approximations, depending on the relative distances between the light source, aperture, and observation screen. In this report, we examine diffraction using the Fresnel approximation, where the observation screen is close enough to the aperture that near-field effects are significant. This allows us to study how the diffraction pattern varies with parameters such as aperture size and screen distance.

To calculate the resulting electric field distribution from an aperture, we use a Fresnel diffraction integral that involves two-dimensional numerical integration. The two techniques used to solve these integrals are Gaussian quadrature, using SciPy's dblquad function, and Monte Carlo integration. The SciPy funciton cannot handle complex function directly, so the integral is decomposed into real and imaginary components using Euler's formula, which relates the complex exponential function to the trigonometric functions cosine and sine. The two options for numerical integration display different advantages, Gaussian quadrature offers high accuracy and efficiency for functions with smooth boundaries, while the Monte Carlo method, based on random sampling, are advantageous when dealing with complex or irregularly bounded apertures.

The objectives of this report are to implement these numerical integration methods for the Fresnel diffraction integral and compare them, to analyze diffraction patterns along a cross-section of the screen and to investigate how aperture size and screen distance influence the observed diffraction pattern. The results provide insights into the effectiveness of different numerical integration techniques in handling complex diffraction integrals and deepen our understanding of near-field diffraction behavior.

2 Theory

The study of diffraction patterns resulting from a light wave passing through an aperture can be analyzed using Fresnel diffraction theory, where the resulting intensity pattern varies based on the distance between the aperture and the observation screen. According to Fresnel's approximation, when light waves propagate parallel to the z-axis and encounter an aperture, they produce a pattern on the screen that varies with z, the distance to the screen. The electric field at a point (x, y, z) on the screen due to a source in the aperture can be calculated as:

$$E(x,y,z) = \frac{e^{ikz}}{i\lambda z} \int\int_{Aperture} \exp\left(\frac{ik}{2z}\left((x-x')^2 + (y-y')^2\right)\right) dx'\, dy' \tag{1}$$

Here $k = 2\pi/\lambda$ represents the wavenumber for the electric waves being diffracted. The double integral is being done with respect to the aperture dimensions dx' and dy', however this calculates the electric field at a point on the screen, as each point in the aperture will contribute to the electric field at every point on the screen. The intensity is proportional to $|E^2|$:

$$I(x,y,z) = \epsilon_0 c E(x,y,z) E^*(x,y,z) = \epsilon_0 c \left|\mathrm{Re}(E(x,y,z))^2 + \mathrm{Im}(E(x,y,z))^2\right| \tag{2}$$

Here ϵ_0 represents the permittivity of free space and c represents the speed of light waves. This equation calculates intensity at one specific point on the screen $I(x, y, z)$ from the real and imaginary components of electric field strength at that point.

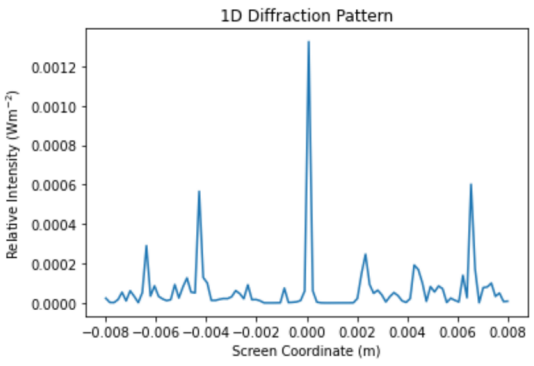
3 Methods and Findings

3.1 Part 1: 1D Diffraction from 2D Aperture

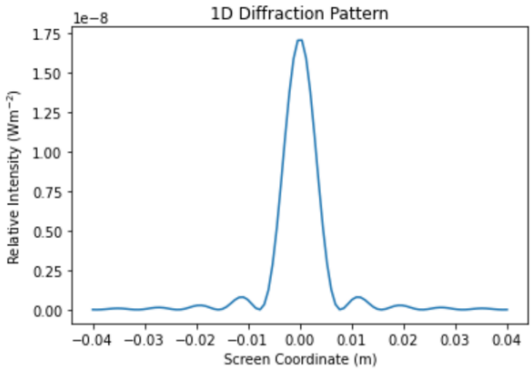
For modeling a cross section of intensity at the screen the integral is evaluated at each point across an array of x values spanning the screen dimensions, keeping y at zero to achieve a 1D diffraction pattern. Gaussian quadrature is the integration method used here to approximate the integral by summing weighted values of the function at specified points. For this the function scipy.integrate.dblquad() in Python is used to compute each part of the integral, split into real and imaginary components, over the aperture region.

The resulting solutions are used to calculate intensity by equation (2) and plotted as magnitude of intensity against distance along the aperture.

Fresnel (near-field) and Fraunhofer (far-field) diffraction

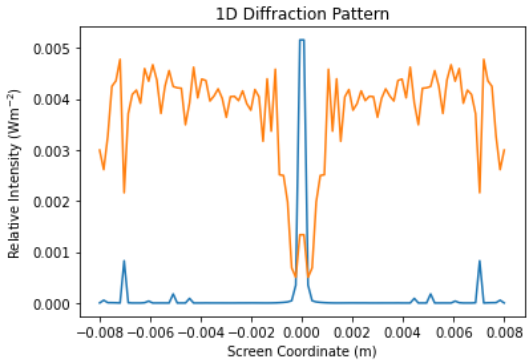


(a) Fresnel near-field diffraction (z = 0.005 m, Aperture width = 2e-4m)

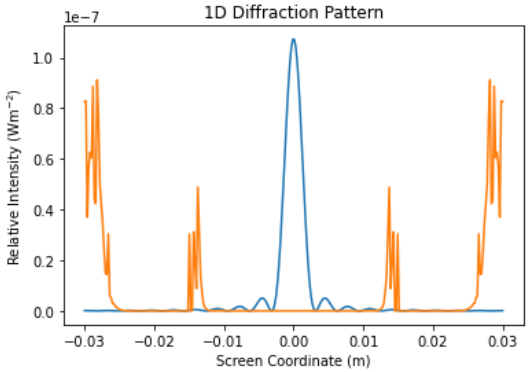


(b) Fraunhofer far-field diffraction (z = 0.05 m, Aperture width = 2e-5m)

Accuracy decreases with distance away from the center of the screen for far field diffraction, this can be changed by specifying epsabs and epsrel, representing absolute and relative tolerance.

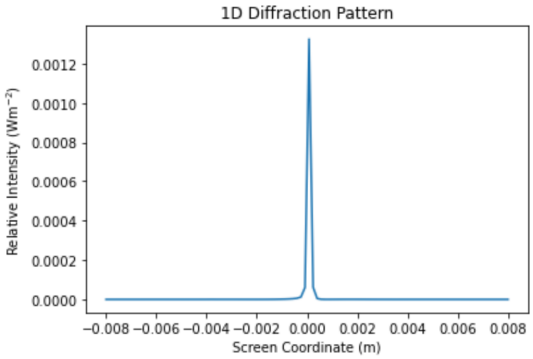


(a) Fresnel near-field diffraction in blue with error in orange

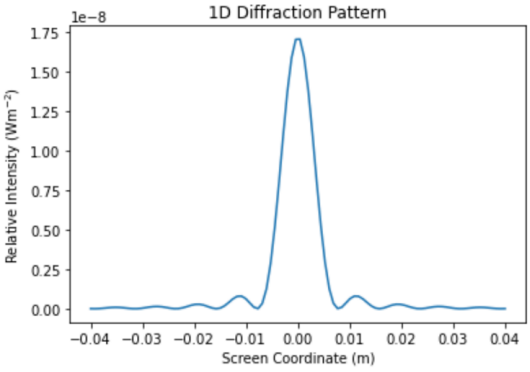


(b) Fraunhofer far-field diffraction in blue with error in orange

Default values of epsabs amd epsrel are $1.49 \cdot 10^{-8}$, by decreasing this to $1 \cdot 10^{-10}$ the error in intensity decreases, leaving a more accurate plot of diffraction.



(a) Fresnel near-field diffraction with lowered tolerance



(b) Fraunhofer far-field diffraction with lowered tolerance

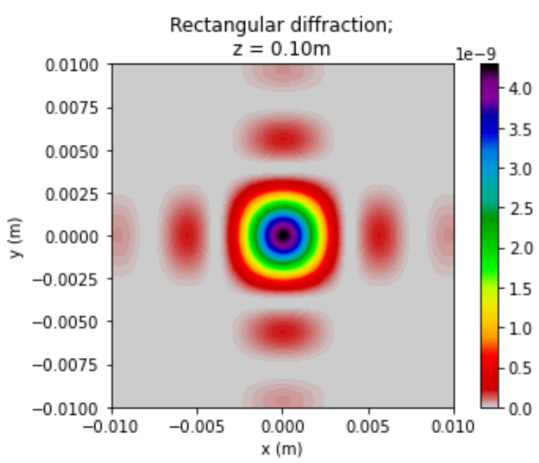
It can be seen that accuracy is more of a problem with larger apertures, in near field diffraction when z is small and further from the center of the screen along the axis. This can be due to several problems, firstly a large aperture increases the integration range, making the phase variations more complex. These variations cause the integrand in the Fresnel diffraction integral to oscillate rapidly, which numerical integration methods struggle to handle without significantly increasing the number of sample points. This also results in higher computational demands, making an inverse relation between accuracy and performance.

The near-field diffraction being less accurate is due to small distances leading to significant phase variations due to the quadratic phase term in the Fresnel approximation. This causes the same rapid oscillations as from a large aperture.

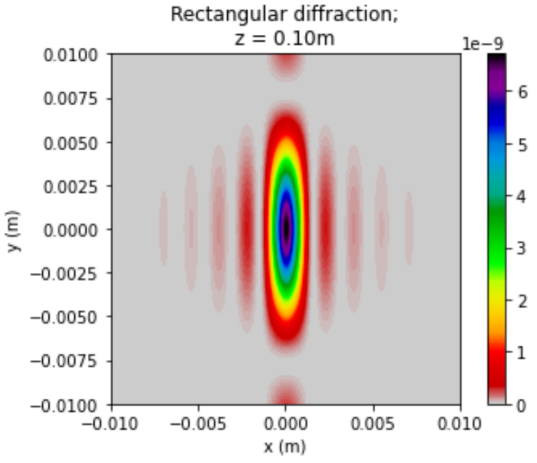
Error increases at points farther from the optical axis because the path length differences across the aperture become larger, leading to even greater phase variations. This amplifies oscillations in the integrand and will cause numerical instability as in the other two sources of error.

3.2 Part 2: 2D diffraction from 2D rectangular aperture

In this section the visualization of a 2D diffraction pattern resulting from a rectangular aperture will be coded by solving the Fresnel diffraction equation over a defined range. Intensity will be stored in a $2D$ array in contrast to the $1D$ array from part 1. The effect of varying aperture size and screen distance on the resulting pattern will also be explored. This model will provide insights into diffraction behavior under different physical conditions, highlighting qualitative differences in patterns for various aperture shapes and screen distances.



(a) Far field diffraction intensity from a square aperture



(b) Far field diffraction intensity from a rectangular aperture

By changing the shape of the aperture to a rectangle the diffraction pattern shifts accordingly, appearing to squash itself to the shape of the aperture the field is diffracted by.

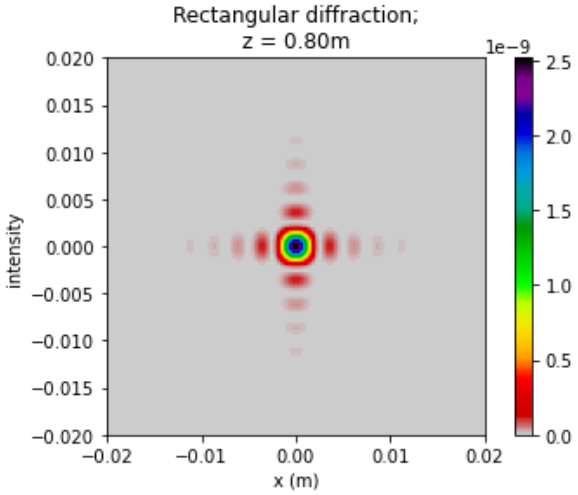


Figure 5: far-field diffraction with 0.8m from aperture to screen

By increasing z, the distance from the aperture to the screen, the intensity drops in magnitude, this aligns with expectation as wave intensity has an inverse square relation with distance from the source.

3.3 Part 3: Part 3: 2-d diffraction from 2-d circular aperture

Here the aperture shape is circular, this causes the limits of the Fresnel diffraction integral to be dependent on each other. Limits of the inner integral are input as a function of the limits of the outer integral, meaning as x' evolves linearly along the aperture y' will change non-linearly, drawing out a circular relation between them.

The relation between y' and x' is $y' = \sqrt{R^2 - x'^2}$, this is coded as a function for the top and bottom integral limits.

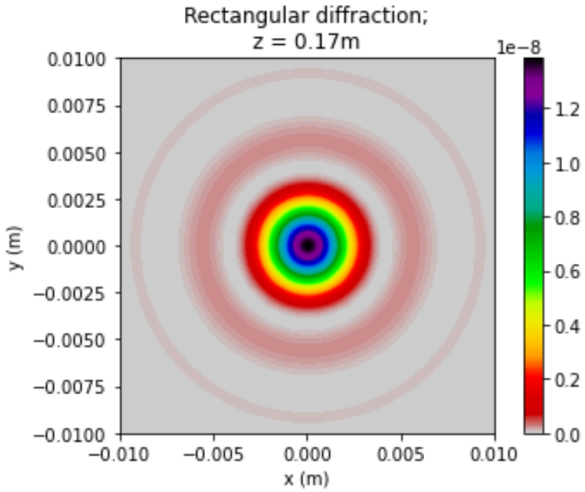
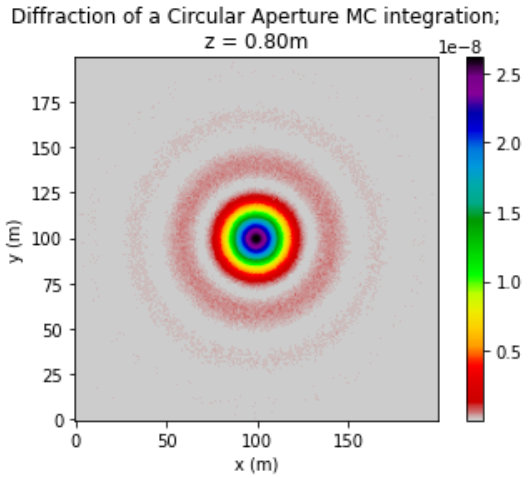


Figure 6: Far field diffraction pattern with a circular aperture

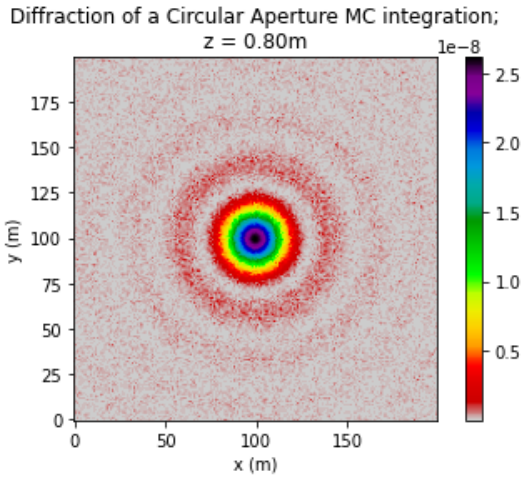
3.4 Part 4: 2D circular diffraction using Monte Carlo

Here Monte Carlo integration is applied to solve the circular diffraction problem and generate a 2D diffraction pattern, in contrast to what was done in part 3 using the 'scipy.integrate.dblquad()' function. The MC method approximates the integral of the Fresnel diffraction equation over a circular aperture by using random sampling across the enclosing square, setting the function value to zero for points outside the circle.

The integral is approximated as an average of the Fresnel expression evaluated at random points within the circle. This approach may be less accurate than exact integration for more simple shapes, however it allows for flexibility and simplicity in handling complex regions to integrate over which may require long winded limits. The electric field and corresponding intensity values across the screen will be calculated, the accuracy will depend mainly of the sampling density.



(a) Far field diffraction from monte carlo integration with 3000 samples



(b) Far field diffraction from monte carlo integration with 300 samples

As the number of samples increases the diffraction pattern becomes clearer, mirroring the intensity plot from part 3 more closely. As error has the number of random samples as it's denominator the resulting intensity plot will become more precise by increasing sample number, this is also shown by

the grainy quality of the plot with fewer samples.

Although the monte carlo integration method is less accurate, it required far less computing power by using averaging multiplied with area rather than solving two double integrals for each point on the screen, as the Gaussian quadrature method uses in part 3. This means depending on the complexity of the area being integrated over, and therefore the computing power required to solve the integrals, the monte carlo method may be advantageous.

4 Appendix

Word Count: 775

4.1 Part 1: 1D Diffraction from 2D Aperture

```
def Fresnel2dreal(yp, xp, y, x, k, z): # real part

    return np.cos((k/(2*z)) * ( (x-xp)**2 + (y-yp)**2 ))


def Fresnel2dimag(yp, xp, y, x, k, z): # imaginary part

    return np.sin((k/(2*z)) * ( (x-xp)**2 + (y-yp)**2 ))
```

Figure 8: Functions for the real and imaginary function to be integrated using scipy.integrate.dblquad()

```
# array set up

realparts = np.zeros(num) # to store the real part of the solved double integral
imagparts = np.zeros(num)

I_xarray = np.zeros(num) # to store the array of intensity values along the cross section

real_error = np.zeros(num) # to store real error in the integral
imag_error = np.zeros(num) # to store imaginary error in the integral

err = np.zeros(num) # to store error in intensity across the cross section
```

Figure 9: Empty arrays set up to store information for each point on the screen across the cross section. The length of each array is num, being the length of xarray specifying uniformly distributed points across the cross section of the screen.

```
y = 0 # 1-d diffraction pattern is cross section of intensity at a specific y value,

xarray = np.linspace(-8e-3, 8e-3, num) # array of x values to solve double integral to


for i in range(num): # loop for every data point

    x = xarray[i] # running through xarray setting x to each value in turn
    realpart, realerr = dblquad(Fresnel2dreal, xp1, xp2, yp1, yp2, args=(y, x, k, z)) # solving integral with the specific x value
    realparts[i] = realpart  # Store the result in thearray realparts

    real_error[i] = realerr

    imagpart, imagerr = dblquad(Fresnel2dimag, xp1, xp2, yp1, yp2, args=(y, x, k, z))
    imagparts[i] = imagpart # Store the result in the array imagparts
    imag_error[i] = imagerr

    #use these to calculate intensity at each x value and store in array I_xarray

    I = ep0*c*((coeff*realpart)**2 + (coeff*imagpart)**2) # intensity equation from real and imaginary parts
    I_xarray[i] = I # storing intensity values


    err_ = ep0*c*((coeff*realerr)**2 + (coeff*imagerr)**2) # intensity equation from real and imaginary parts
    err[i] = err_
```

Figure 10: Keeping y constant to achieve the desired cross section across the x axis along the screen, the x value is looped across the whole array of 'xarray' to calculate intensity at each point across the screen. These values are stored and converted to intensity to plot.

4.2 Part 2: 2D diffraction from 2D rectangular aperture

The code here is similar to part 1, with the same 'Fresnel2dreal' and 'Fresnel2dimag' functions. The exercises differ in the x and y values both varying here to achieve a 2D array of intensity for the screen.

```
# empty arrays to store results of integral for the array of x and y values
real_parts = np.zeros((num,num))
imag_parts = np.zeros((num,num))
intensity = np.zeros((num,num))

# array of x and y values to integrate with
xarray = np.linspace(-4e-2, 4e-2, num) # array of x values to solve double integral to
yarray = np.linspace(-4e-2, 4e-2, num) # array of y values to solve double integral to
```

Figure 11: Setting up empty arrays to store real and imaginary components of electric field strength at the screen to be combined by equation (2), storing the resulting intensity at each point in another 2D array. Uniform arrays of x and y spanning the aperture are set up to grid the screen into points to evaluate intensity at. These all have length 'num' to ensure the resulting intensity array is the same size as the screen, each point having a calculated intensity to be plotted.

```
# main code

for i in range(num): # loop for every data point

    x = xarray[i] # running through xarray setting x to each value in turn

    for j in range (num):

        y = yarray[j]

        # solving integral for array of y values in yarray[j] at specifc x value xarray[i]
        # storing results in array where real_parts[:,0] is from array of x values and real_parts[:,1] is from array of y values

        real_parts[i,j], realerr = dblquad(Fresnel2dreal, xp1, xp2, yp1, yp2, args=(y, x, k, z))

        # repeat for imaginary part of the equation

        imag_parts[i,j], imagerr = dblquad(Fresnel2dimag, xp1, xp2, yp1, yp2, args=(y, x, k, z))

        # solve for intensity and store into an array

        intensity[i,j] = ep0*c*((coeff*real_parts[i,j])**2 + (coeff*imag_parts[i,j])**2)
```

Figure 12: Looping both x and y values to evaluate the diffraction integral at each point on the screen, storing the real and imaginary components of the solved integral and combining them into the intensity array.

```
extents = (x1,x2,y1,y2) # Sets the limits for the plot
plt.imshow(intensity,vmin=0.0,vmax=1.0*intensity.max(),extent=extents,\
           origin="lower",cmap="nipy_spectral_r") # Try different
plt.xlabel("x (m)")
plt.ylabel("y (m)")
plt.title('Rectangular diffraction;\nz = {:4.2f}m'.format(z))
plt.colorbar()
plt.show()
```

Figure 13: Plotting intensity using 'plt.imshow()', which renders 2D scalar arrays as intensity has been stored in as a pseudocolour image. Increasing 'num' increases the number of pixels the image is rendered with, increasing the precision of the plot.

Figure 14: Colour map used in 'plt.imshow()', where the far left maps to the largest scalal data points.

4.3 Part 3: 2D diffraction from 2-d circular aperture

In this exercise two new functions are added to the code, still using Gaussian quadrature to solve the Fresnel diffraction integral.

```
def Fresnel2dreal(yp, xp, y, x, k, z): # real part of inner integral
    return np.cos((k/(2*z)) * ( (x-xp)**2 + (y-yp)**2 ))

def Fresnel2dimag(yp, xp, y, x, k, z): # imaginary part
    return np.sin((k/(2*z)) * ( (x-xp)**2 + (y-yp)**2 ))

def yp1func(xp): # function for inner y integral lower limit
    return - np.sqrt(R*R - xp*xp)
def yp2func(xp): # function for inner y integral upper limit
    return np.sqrt(R*R - xp*xp)
```

Figure 15: Here inner integral limits for and y' are input as functions to integrate over a circular aperture of radius R.

```
for i in range(num): # loop for every data point
    x = xarray[i] # running through xarray setting x to each value in turn
    for j in range (num):
        y = yarray[j]
        # solving integral for array of y values in yarray[j] at specifc x value xarray[i]
        # storing results in array where real_parts[:,0] is from array of x values and real_parts[:,1] is from array of y values
        real_parts[i,j], realerr = dblquad(Fresnel2dreal,xp1,xp2,yp1func,yp2func,args=(y, x, k, z), epsabs=1e-10, epsrel=1e-10)
        # repeat for imaginary part of the equation
        imag_parts[i,j], imagerr = dblquad(Fresnel2dimag,xp1,xp2,yp1func,yp2func,args=(y, x, k, z), epsabs=1e-10, epsrel=1e-10)
        # solve for intensity and store into an array
        intensity[i,j] = ep0*c*((coeff*real_parts[i,j])**2 + (coeff*imag_parts[i,j])**2)
```

Figure 16: Loop of x and y values across the screen as in part 2, differing in the y_1' and y_2' values in the integral being replaced with functions shown in figure 14.

Plotting used the same 'plt.imshow()' as part 3, as intensity is again stored as a 2D scalar array.

4.4 Part 4: 2D circular diffraction using Monte Carlo

```
def monte(Fresnel2dreal,Fresnel2dimag,xp1,xp2,N_samples, _x, _y, R):
```

Figure 17: New function 'monte' defined for part 4, its function is to calculate intensity at a specified point on the screen using monte carlo integration rather than Gaussian quadrature.

```
# mask set up

samples_xp = np.random.uniform(low=xp1,high=xp2,size=N_samples) # random x values of samples on the aperture
samples_yp = np.random.uniform(low=yp1,high=yp2,size=N_samples) # random y values of samples on the aperture
samples_rp = np.sqrt( samples_xp**2 + samples_yp**2 ) # array of displacements of samples from the origin

mask = samples_rp < R # mask for all sample points within circular aperture

N_mask = sum(mask) # now there are N_mask number of sample points we will use, not N_samples

good_samples_xp = samples_xp[mask] # all values in samples_rp which align for TRUE in mask boolian will be taken into good_samples_xp
good_samples_yp = samples_yp[mask]

# now only points within circle are passed through function so no need for a loop 'if', 'else' later
```

Figure 18: Within the monte function "N samples" number of random sample points are spread across a square area enclosing the circular aperture. A mask is set up to filter out the points outside of the circular shape, this is done by calculating the displacement of each point and wiping out points where this displacement is greater than the radius R of the aperture.

```
# sum of the real and imaginary parts of the function within the electric field integral
real_values_sum = sum(Fresnel2dreal(good_samples_yp, good_samples_xp, _y, _x, k, z))
imag_values_sum = sum(Fresnel2dimag(good_samples_yp, good_samples_xp, _y, _x, k, z))

real_mean = real_values_sum / N_mask # mean of real parts of function across all sample points
imag_mean = imag_values_sum / N_mask # mean of imaginary parts of function across all sample points

area_aperture = np.pi*R**2 # m^2, area of the aperture

real_E = coeff*area_aperture*real_mean # real parts of electric field following the MC integral formula
imag_E = coeff*area_aperture*imag_mean # imaginary parts of electric field following the MC integral formula

intensity = ep0*c*( real_E**2 + imag_E**2 ) # intensity calculation
```

Figure 19: Within monte function the mean of the function $< f >$ in the Monte Carlo integration formula is split into real and imaginary mean, each of these are calculated by summing the real and imaginary components of the function. These sums are divided by the number of samples within the circle to get the mean of both real and imaginary components. This combined with the area of the aperture are used to calculate intensity.

```
# error calculation

squared_real_values_sum = sum((Fresnel2dreal(good_samples_yp, good_samples_xp, _y, _x, k, z))**2)
squared_imag_values_sum = sum((Fresnel2dimag(good_samples_yp, good_samples_xp, _y, _x, k, z))**2)

meansq = (squared_real_values_sum + squared_imag_values_sum) / N_mask

# done for intensity, cut out middle man of electric field array, now compute <f**2> from real mean and imag mean

error = (area_aperture) * np.sqrt(( meansq- (real_mean**2 + imag_mean**2) ) / N_mask )
```

Figure 20: Error is carried through the same calculations to get error in the resulting intensity.

```
# array set up

intensity_array = np.zeros((N,N)) # empty array to store results of integral for the array of x and y values
error_array = np.zeros((N,N))

xarray = np.linspace(-4e-2, 4e-2, N) # array of x values to evaluate the Fresnel expression to
yarray = np.linspace(-4e-2, 4e-2, N) # array of y values to evaluate the Fresnel expression to
```

Figure 21: Set up of arrays to store intensity and error in intensity, x and y arrays that span the screen.

```
for i in range(N): # double loop for every point on the screen from array of x and y values

    x = xarray[i]

    for j in range(N):

        y = yarray[j]

        # for point on screen (xarray[i], yarray[j]):

        intensity , error = monte(Fresnel2dreal,Fresnel2dimag,xp1,xp2,N_samples, x, y, R)
        intensity_array[i,j] = intensity
        error_array[i,j] = error
```

Figure 22: Double loop for each point on the screen the monte function is called to solve for intensity and error in intensity for that point, this is stored in arrays to plot.