Exercise 2 Report

Ceres Milner[1]

[1] *Department of Physics, University of Bristol*

01.12.2025

Word count: 1741

Abstract

This exercise aimed to use numerical integration methods to investigate diffraction of light through an aperture onto a screen. Two calculation methods were used, a numerical calculation and Monte Carlo integration. The numerical method gave cleaner, less 'noisy' data, but the Monte Carlo method was considerably quicker.

1 Introduction

In this exersise we will use numerical methods to investigate the diffraction of light when passed through an aperture of different shapes. We will use two methods, first pure numerical integration methods, and then the Monte Carlo method, and compare the accuraccy and speed of these two methods. We will first create a 1 dimensional plot of the diffraction to ensure the results are as expected, and then 2 dimensional plots, using both square and curcular apertures.

2 Theory and Methods

To calculate the diffraction pattern of light passing through a single aperture, we will use the Fresnel diffraction integral, given by:

$$E(x,y,z) = \frac{i^{ikz}}{i\lambda z}\int_{-\infty}^{\infty}\int_{-\infty}^{\infty} E(x',y')\exp\left\{\frac{ik}{2z}\left[(x-x')^2+(y-y')\right]\right\}\mathrm{d}x'\mathrm{d}y' \tag{1}$$

To allow for this to be calculated compuptionally, we will simplify it, integrating over the area of the aperture, and seperating out the real and imaginary parts of the equation:

$$\begin{aligned} E(x,y,z) = &\frac{kE_0}{2\pi z}\int_{x_1'}^{x_2'}\int_{y_1'(x')}^{y_2'(x')}\cos\left\{\frac{k}{2z}\left[(x-x')^2+(y-y')^2\right]\right\}\mathrm{d}x'\mathrm{d}y' \\ &+i\frac{kE_0}{2\pi z}\int_{x_1'}^{x_2'}\int_{y_1'(x')}^{y_2'(x')}\sin\left\{\frac{k}{2z}\left[(x-x')^2+(y-y')^2\right]\right\}\mathrm{d}x'\mathrm{d}y' \end{aligned} \tag{2}$$

3 Explanation of Code

Initially, the functions for the kernels of the real and imaginary parts of equation 2 are set up, as well as setting some universal constants:

```
def Fresnel2dreal(yp, xp, y, x, k, z): #Define functions for integral kernels
    kernel = np.cos((k/(2*z))*((x-xp)**2+(y-yp)**2))
    return kernel

def Fresnel2dimag(yp, xp, y, x, k, z):
    kernel = np.sin((k/(2*z))*((x-xp)**2+(y-yp)**2))
```

```
    return kernel

c = 3e8 #Universal constants
e0 = 8.85e-12
```

A function for part 1 is then defined:

```
def plot1D(aperture, z, k, screen_range, resolution): #Function for part 1
    def genData(aperture, z, k, screen_range, resolution): #Function to generate data
        y = 0 #As in 1D

        xp1=yp1=-aperture/2 #Set range over which we are integrating
        xp2=yp2=aperture/2

        xs = np.linspace(-screen_range/2, screen_range/2, num=resolution) #Generate values to
            calculate for
        intensities = []

        constant = k/2*np.pi*z #Relative intensity constant

        for x in tqdm(xs): #Loop through all values of x
            realpart, realerror = integrate.dblquad(Fresnel2dreal, xp1, xp2, yp1, yp2, args=(y, x,
                k, z), epsabs=1e-10, epsrel=1e-10) #Calculate both real and imaginary parts
            imagpart, imagerror = integrate.dblquad(Fresnel2dimag, xp1, xp2, yp1, yp2, args=(y, x,
                k, z), epsabs=1e-10, epsrel=1e-10)

            I = c*e0*((realpart*constant)**2+(imagpart*constant)**2) #Combine parts and constants to
                get intensity
            intensities.append(I) #Add calculated intensity to list
        return xs, intensities

    ax = plt.axes()
    xs, intensities = genData(aperture, z, k, screen_range, resolution) #Plot intensity against
        distance
    ax.plot(xs, intensities)
    plt.xlabel("Position (m)")
    plt.ylabel("Relative Intensity")
    plt.title("1D diffraction")
    plt.show()
```

A sub function is defined with inputs of aperture, z, k screen range and resolution, to generate the data to plot. It begins by setting y=0 as it generates a 1D plot, and then setting the limits of the integration, defined by the aperture. An array of x values is then created, and these are the positions on the screen which we will calculate the integral for. The function then iterates through this array, and for each value of x it uses the dblquad function to calculate the real and imaginary part of the integral for the given x value. It then takes these results and combines them with the universal constants to give the intensity at that point, and then appends this value to a list of intensities. This genData function is then called and the data is saved to a variable, and the data is then plotted, creating a graph of intensity against position.

The function for part 2 is then defined:

```
def plot2Drectangular(aperture, z, k, screen_range, resolution): #Function for part 2
    def genData(aperture, z, k, screen_range, resolution): #Function to generate data
        xp1=yp1=-aperture/2 #Set integration limits
        xp2=yp2=aperture/2
```

```
        xs = np.linspace(-screen_range/2, screen_range/2, num=resolution) #Generate values to
            integrate for
        ys = np.linspace(-screen_range/2, screen_range/2, num=resolution)

        intensities = []

        constant = k/2*np.pi*z #Relative intensity constant
        completion = 0

        for y in tqdm(ys):
            xIntensities = []
            for x in xs:
                realpart, realerror = integrate.dblquad(Fresnel2dreal, xp1, xp2, yp1, yp2, args=(y,
                    x, k, z), epsabs=1e-10, epsrel=1e-10)#Calculate both parts
                imagpart, imagerror = integrate.dblquad(Fresnel2dimag, xp1, xp2, yp1, yp2, args=(y,
                    x, k, z), epsabs=1e-10, epsrel=1e-10)

                I = c*e0*((realpart*constant)**2+(imagpart*constant)**2)#Combine both parts and
                    constants
                xIntensities.append(I)
            intensities.append(xIntensities)
        intensities = np.array(intensities)
        return intensities

    intensity = genData(aperture, z, k, screen_range, resolution) #Generate the data
    extents = (-screen_range/2,screen_range/2,-screen_range/2,screen_range/2) #Set limits of the
        plot

    plt.imshow(intensity,vmin=0.0,vmax=1.0*intensity.max(),extent=extents,\
   origin="lower",cmap="nipy_spectral_r") #Plot the graphs
    plt.colorbar()
    plt.xlabel("X Position (m)")
    plt.ylabel("Y Position (m)")
    plt.title("2D diffraction with a rectangular aperture")
    plt.show()
```

The function for the second part is quite similar to the first part. Again, a function to generate the data is defined. It sets the linits for the integration, and this time generates an array of values for both x and y. It then loops through both the y and x lists, with the x loop nested inside the y loop. Each iteration it calculates the integral for the current location, and adds it to a 2D array representing the positions on the screen. Each row of pixels is a list, and the 2D array is a list of these rows. The genData function is then called and its result is saved to intensities, and the extents of the 2D plot are set to the size of the screen. The plot is then generated, with the scale set to be from 0 to the maximum value in the intensities array.

The function for part 3 is then defined:

```
def plot2Dcircular(aperture, z, k, screen_range, resolution): #Function for part 3
    def genData(aperture, z, k, screen_range, resolution):
        xp1=-aperture/2 #Set integration limits
        xp2=aperture/2

        def yp1func(xp):
            return -np.sqrt((aperture/2)**2-(xp**2)) #Define y limits in terms of x

        def yp2func(xp):
            return np.sqrt((aperture/2)**2-(xp**2))
```

```
        xs = np.linspace(-screen_range/2, screen_range/2, num=resolution) #Generate values to
            integrate for
        ys = np.linspace(-screen_range/2, screen_range/2, num=resolution)

        intensities = []

        constant = k/2*np.pi*z

        for y in tqdm(ys):
            xIntensities = []
            for x in xs:
                realpart, realerror = integrate.dblquad(Fresnel2dreal, xp1, xp2, yp1func, yp2func,
                    args=(y, x, k, z), epsabs=1e-10, epsrel=1e-10) #Calculate functions using
                    circular limits
                imagpart, imagerror = integrate.dblquad(Fresnel2dimag, xp1, xp2, yp1func, yp2func,
                    args=(y, x, k, z), epsabs=1e-10, epsrel=1e-10)

                I = c*e0*((realpart*constant)**2+(imagpart*constant)**2)
                xIntensities.append(I)
            intensities.append(xIntensities)
        intensities = np.array(intensities)
        return intensities

    intensity = genData(aperture, z, k, screen_range, resolution)
    extents = (-screen_range/2,screen_range/2,-screen_range/2,screen_range/2)

    plt.imshow(intensity,vmin=0.0,vmax=1.0*intensity.max(),extent=extents,\
   origin="lower",cmap="nipy_spectral_r")
    plt.colorbar()
    plt.xlabel("X Position (m)")
    plt.ylabel("Y Position (m)")
    plt.title("2D diffraction with a circular aperture")
    plt.show()
```

This operates very similarly to the part 2 function. The only difference is the addition of the yp1function and yp2function. These define the y prime limits of the integral in terms of the x prime value, and these are passed to the dblquad functions in order to calculate the integrals using a circular aperture. The function for part 4 is then defined:

```
def monte(aperture, z, k, screen_range, resolution, samples): #Function for part 4
    N = samples #Number of samples

    def doubleInteg(x, y, xp, yp, z, k, aperture): #Function to return calculated values for each
        sample
        values = []
        for i in range(len(xp)):
            if (xp[i]**2+yp[i]**2) > (aperture/2)**2: #Check if point is in the aperture
                values.append(0)
            else:
                value = np.exp(((1j*k)/(2*z))*((x-xp[i])**2+(y-yp[i])**2)) #Calculate value if in
                    aperture
                values.append(value.imag)
        return np.array(values)

    def monteCarlo(x, y, z, k, aperture): #Perform onte carlo method
```

```
    xp = np.random.uniform(low=(-aperture/2), high=aperture/2, size=N) #Generate random samples
    yp = np.random.uniform(low=(-aperture/2), high=aperture/2, size=N)
    values = doubleInteg(x, y, xp, yp, z, k , aperture) #Calculate value for each pair of samples
    mean = values.sum()/N
    meansq = (values*values).sum()/N
    integral = aperture*mean #Calculate the integral end error
    error = aperture*np.sqrt((meansq-mean*mean)/N)
    return integral, error

def genData(aperture, z, k, resolution, screen_range): #Function to generate the data

    xs = np.linspace(-screen_range/2, screen_range/2, num=resolution) #Generate values to
        integrate for
    ys = np.linspace(-screen_range/2, screen_range/2, num=resolution)


    intensities = []

    constant = k/(2*np.pi*z)

    for y in tqdm(ys):
        xIntensities = []
        for x in tqdm(xs):
            integral, error = monteCarlo(x, y, z, k, aperture) #Find integral vaule using monte
                carlo method
            I = c*e0*constant*integral
            xIntensities.append(I)
        intensities.append(xIntensities)
    intensities = np.array(intensities)
    return intensities

intensity = genData(aperture, z, k, resolution, screen_range) #Generate data
extents = (-screen_range/2,screen_range/2,-screen_range/2,screen_range/2)

for y in range(len(intensity)):
    for x in range(len(intensity[y])):
        if intensity[y][x] < 0.05*intensity.max():
            intensity[y][x] = 0

plt.imshow(intensity,vmin=0,vmax=1.0*intensity.max(),extent=extents,\
origin="lower",cmap="nipy_spectral_r")
plt.colorbar()
plt.xlabel("X Position (m)")
plt.ylabel("Y Position (m)")
plt.title("2D diffraction through Monte Carlo")
plt.show()
```

The function doubleInteg is defined. This takes values of x, y, z, k and aperture, as well as a list of random samples of xp and yp. For each value pair of xp and yp, it test if the coordinate is in the circular aperture. If it is, the kernel function is calculated and appended to the array of values, and of the coordinate is outside of the aperture, the value is said to be 0 and this is appended to the list of values. The monteCarlo function is then defined. This first generates N random samples of xp and yp in two arrays. It then calls the doubleInteg function, passing these arrays of samples as well as the other needed values, and saves the resulting list of values to the values variable. From this list of values, it calculates both the mean and mean squared, and uses these to calculate an approximation of the integral, as well as the error associated with this integral, and returns both. As with the previous sections, the

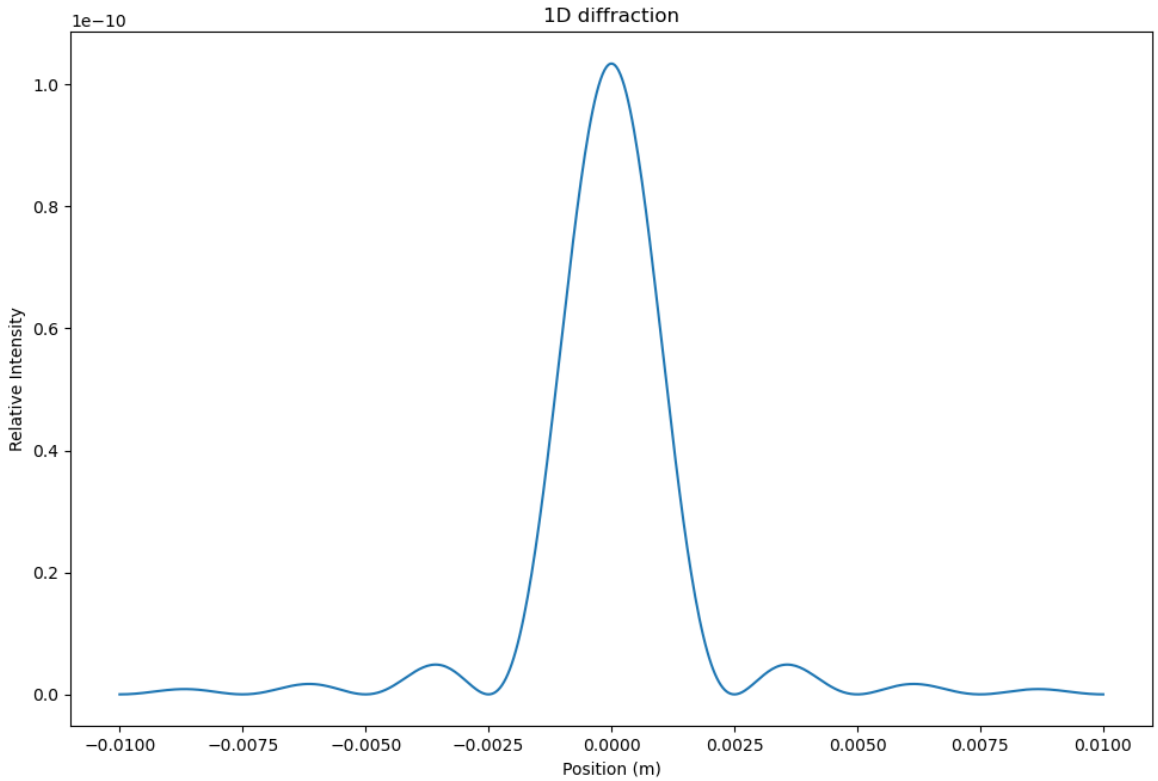

Figure 1: 1D Far-field diffraction

genData function is the defined, first creating arrays of x and y values, then looping through these arrays. This time however, instead of using the dblquad function it uses the monteCarlo function to calculate the integral, then calculates the intensity from this. Again, it saves each value to a 2D array, and returns this array. The genData function is then called and the array is saved to intensity, and the plot extents are set. As the Monte Carlo method generates a lot of low-level noise, the intensity is then looped through and all values less than 5% of the maximum are set to 0. While this may remove some actual data, it greatly increases the signal to noise ration, allowing for the diffraction pattern to be observed more easily. The data is then plotted in the same manner as parts 2 and 3.

4 Results and Discussion

Throughout these results a wavelength of 1e-6m will be used. For far field diffraction, an aperture of 2e-5m and a z of 0.05m will be used, and for near field diffraction an aperture of 2e-4 and a z of 0.005m will be used.

4.1 Section 1: 1D diffraction

The 1D diffraction performs very well for far field diffraction, giving a graph in the expected shape, as shown in figure 1 For near field diffraction, initially the plot was very noisy, with many spikes as shown in figure 2. To correct for this, the epsabs and epsrel arguments to 1e-10, which produced the correct diffraction pattern, as shown in figure 3

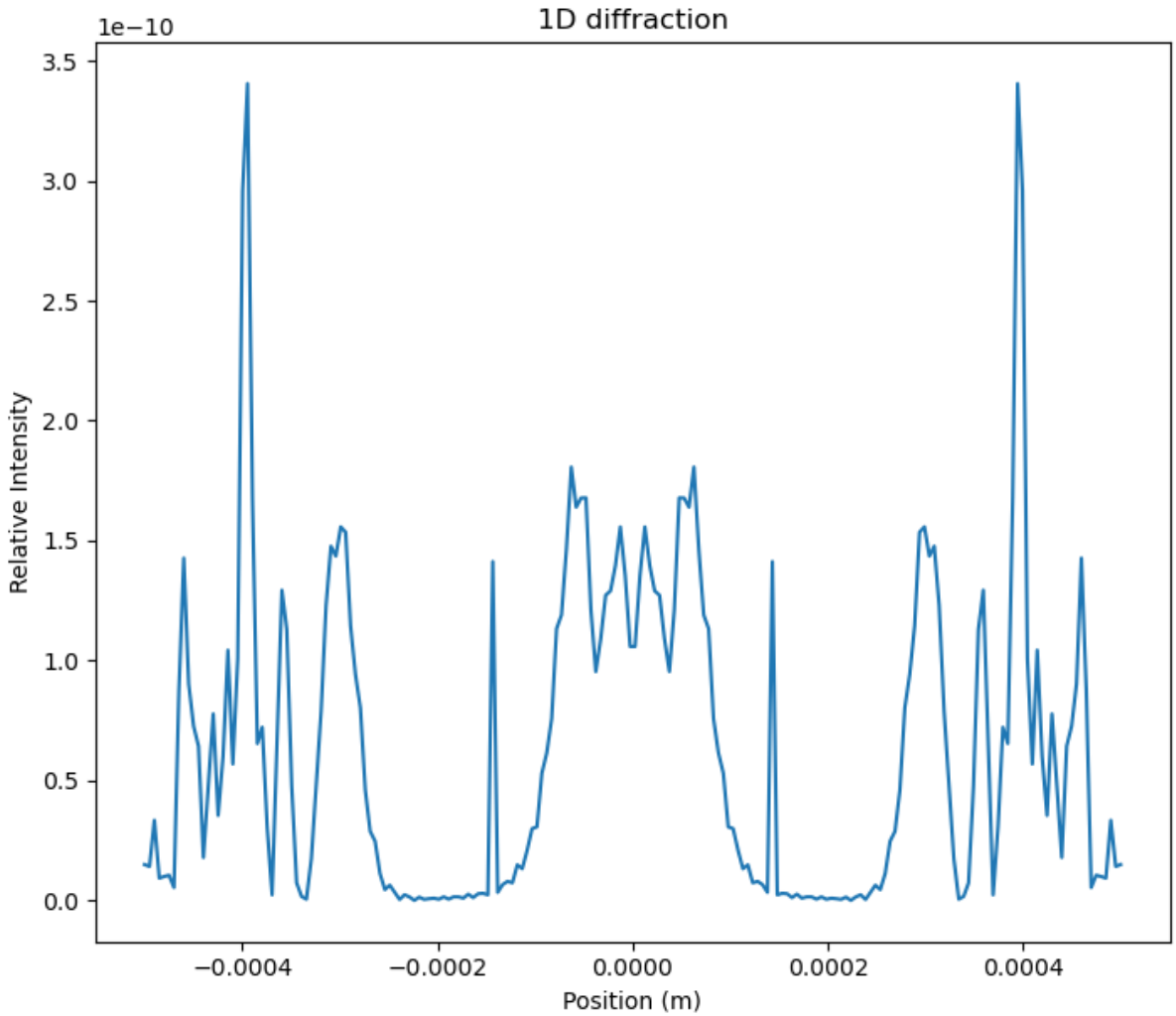

Figure 2: 1D Near-field diffraction with incorrect tolerences

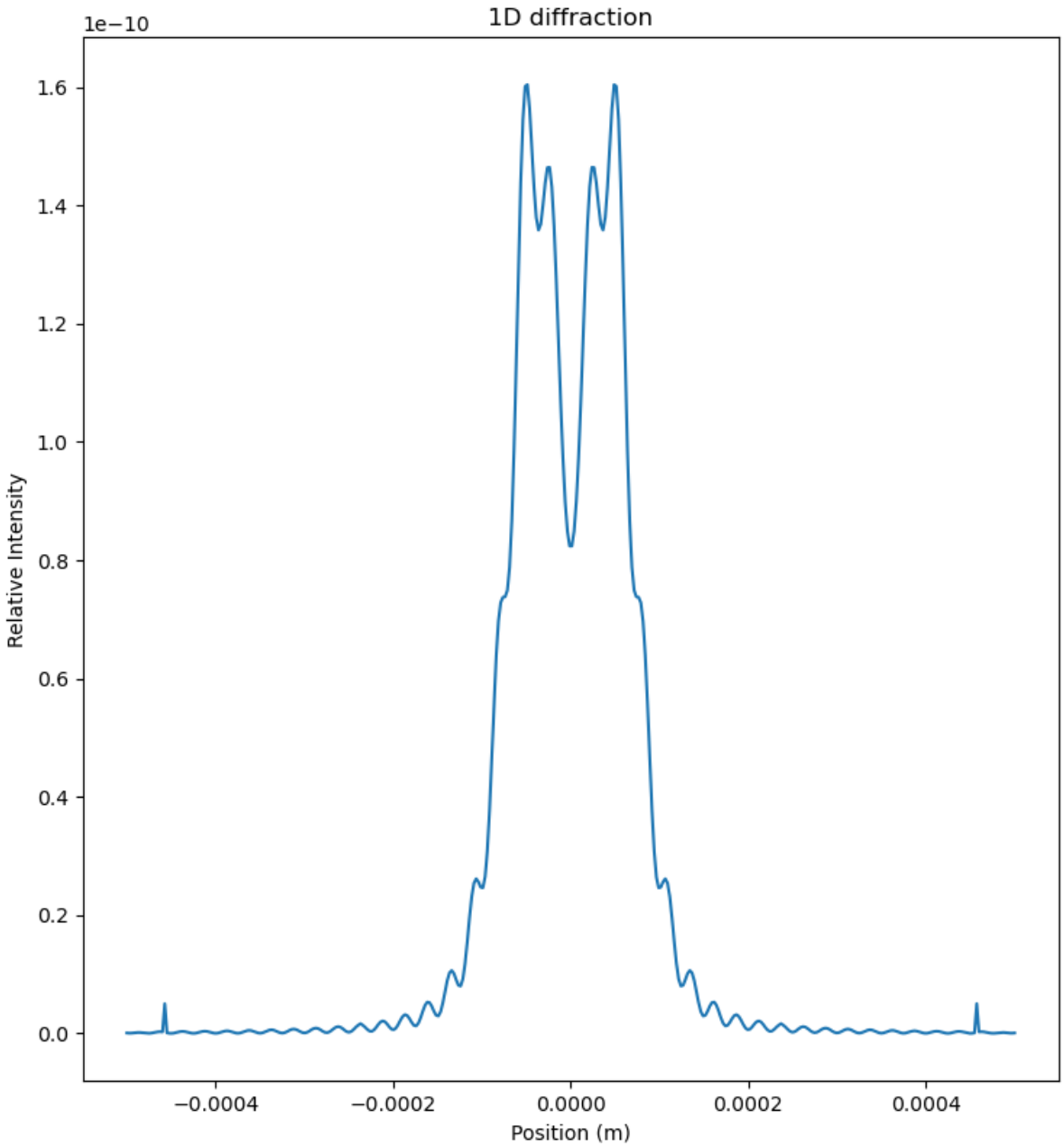

Figure 3: 1D Near-field diffraction

4.2 Section 2: 2D diffraction with a square aperture

Section 2 of the code also behaves as expected. It begins to become quite time intensive at high resolutions due to the large number of integrations needed to be calculated. It produces the expected pattern for both near and far feild diffraction, as shown in figures 4 and 5 As the near field diffraction takes longer to compute than the far field, the near field plot was generated using a lower resolution of 100×100, whereas the far field plot was 400×400.

4.3 Section 3: 2D diffraction with a circular aperture

Again, section 3 gave the expected results, as shown in figures 6 and 7, however for this section the near field diffraction took much longer to compute than the far field, so again it was performed at a lower resolution.

4.4 Section 4: 2D diffraction through Monte Carlo

The Monte Carlo method computed much faster than the numerical integration. At low resolutions and a low number of samples, a weak pattern was observed in the centre of the plot, but the rest of the data was effectively just noise, as shown in figure 8. While this is very quick to compute, it is very hard to draw any data from it. When given a low resolution of 50×50 pixels, but a high sample rate of 5000 samples, the plot was still preduced quite quickly, but now showed a strong pattern, shown in figure 9, although with this low resolution it is still hard to confirm the accuraccy of the pattern. For a plot with a high resolution of 500×500 and 10 samples per pixel, the pattern was more distinguishable than the low resolution plot, but outside the middle of the plot the data quickly became noise and it is hard to make out any pattern, as shown in figure 10 For a plot with both a high resolution of 600×600 pixels and a high number of samples, 1000, a clear pattern is created, show in fgure 11, although this takes much longer to compute than the previous plots. This plot also contains regular circular patterns not present in the numerical integrations. When compared with the numerical method, the MonteCarlo method shows the same pattern, although with more noise present and some irregularities. Unlike the previous sections, the near field pattern does not take much longer than the far field, likely as the dblquad function tests for convergence, which takes longer with the integrations present in the nearfield situation, however this is not the case for the Monte Carlo method, so the near field situation takes aproximately the same time as the far firld situation. When using a high resolution of 400×400 pixels and a high number of scamples of 1000, the Monte Carlo method also produces a clear pattern for the near field situation, shown in figure 12, and again this matches the pattern produced by the numerical integration

4.5 Section 5: Time for data generation

To compare the efficiency of the numerical method and the Monte Carlo method, plots were generated for a number of different resolutions using the numerical method and the Monte Carlo method at a nu,ber of different sample rates. A plot of time against resolution was then created. As the number of integrations performed is proportional to resolution^2, we would expect the time taken to also be proportional to resolution^2. As such, time was plotted against resolution^2 to produce straight lines, shown in figure 13. As expected, each line follows a very strong linear relationship, showing that time is indeed proportional to resolution^2. Additionally, we can see that the samples needed for the Monte Carlo method to match the numerical method in efficiency is approximately 1600, however even at theis number of samples the Monte Carlo method still gives less clear data than the numerical method. As such, to balance if the accuracy of a Monte Carlo plot with samples $\leq$ 1600 is enough, the Monte Carlo method should be used as it is much faster, but if a greater accuracy is needed, the numerical method should be used.

5 Conclusion

All numerical methods gave the expected results, hadeling both near and far field diffraction, however at higher resolutions the 2 dimensional plots began to take a large amount of time to compute. Using the Monte carlo method, if either the resolution of number of samples was too low, the produced plot gave little useful data, however with

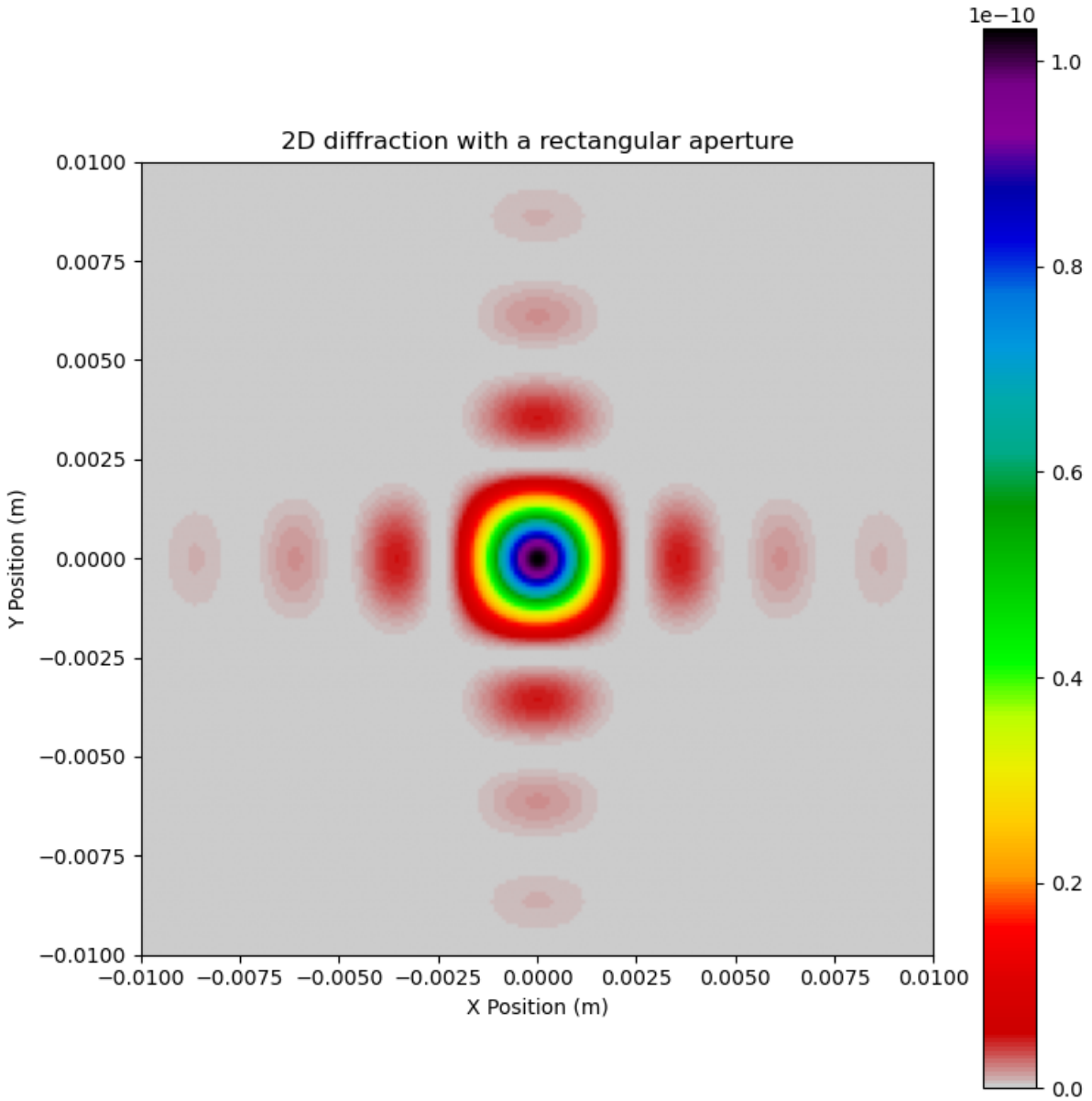

Figure 4: 2D Far-field diffraction

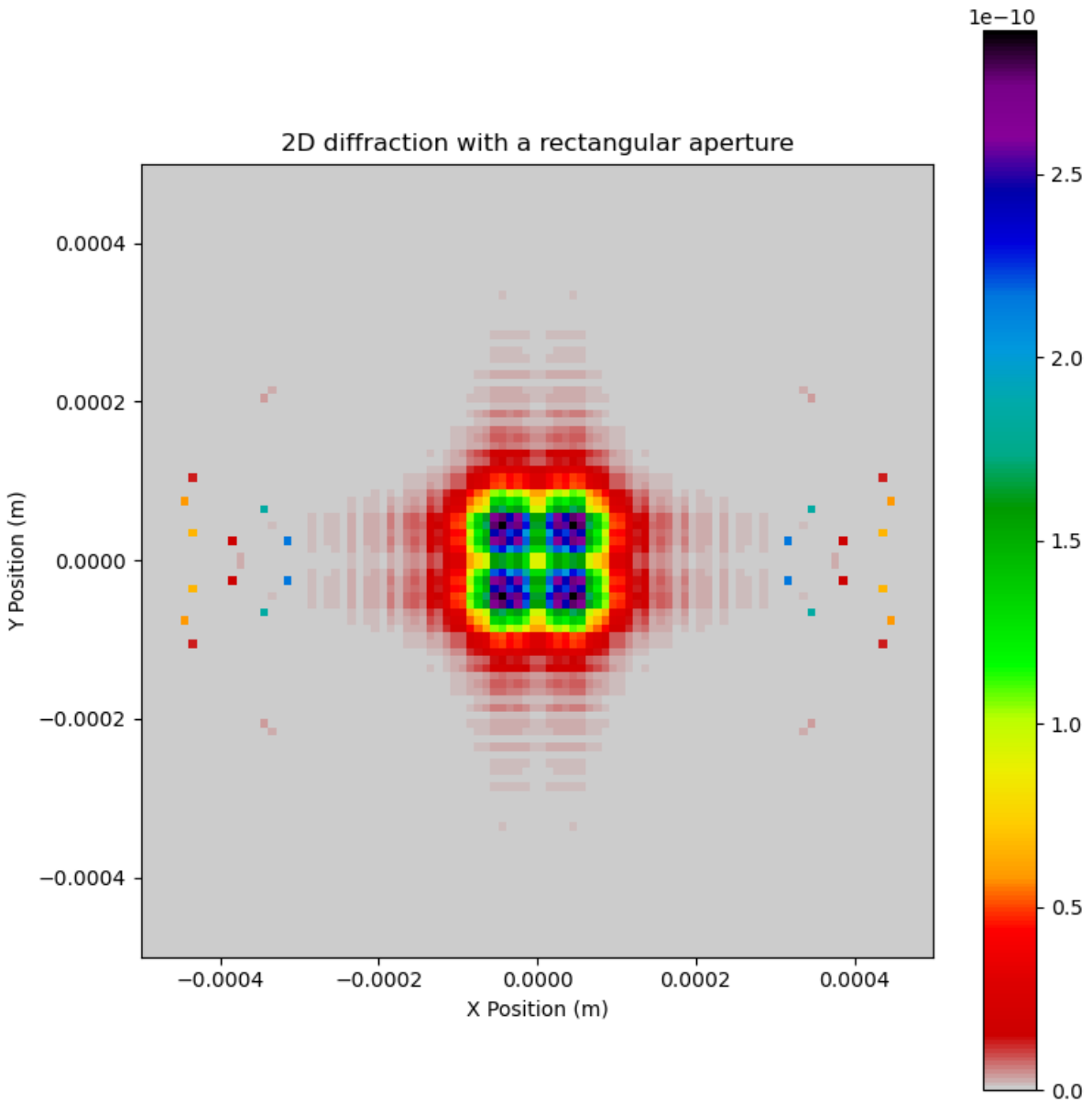

Figure 5: 2D Near-field diffraction

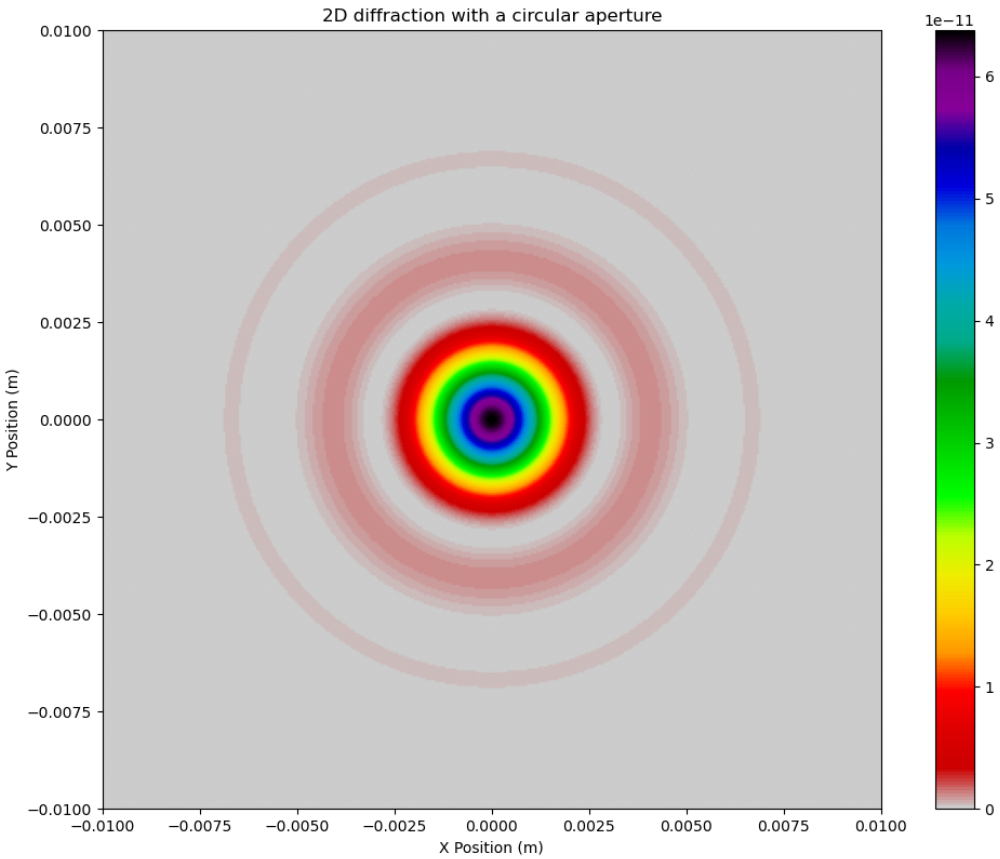

Figure 6: 2D Far-field diffraction with a circular aperture

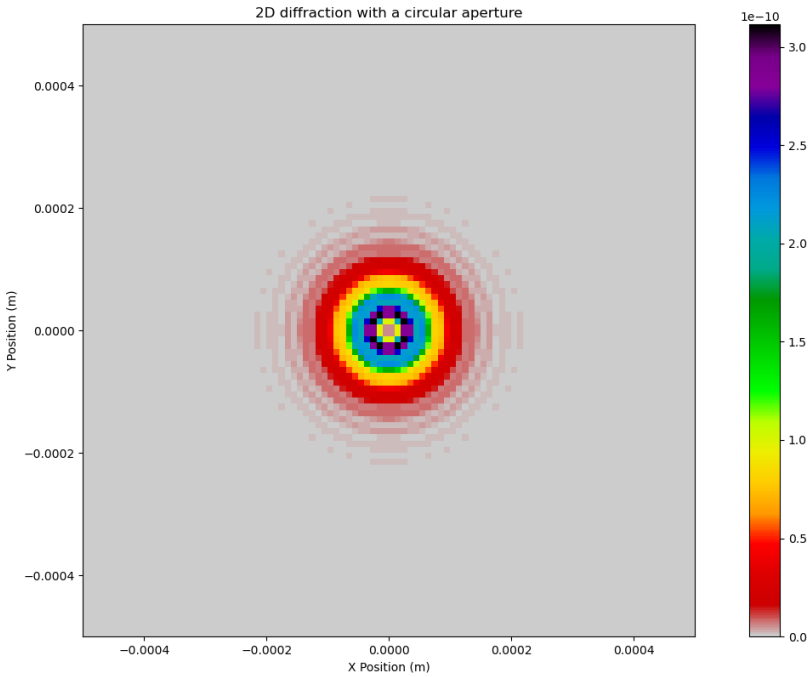

Figure 7: 2D Far-field diffraction with a circular aperture

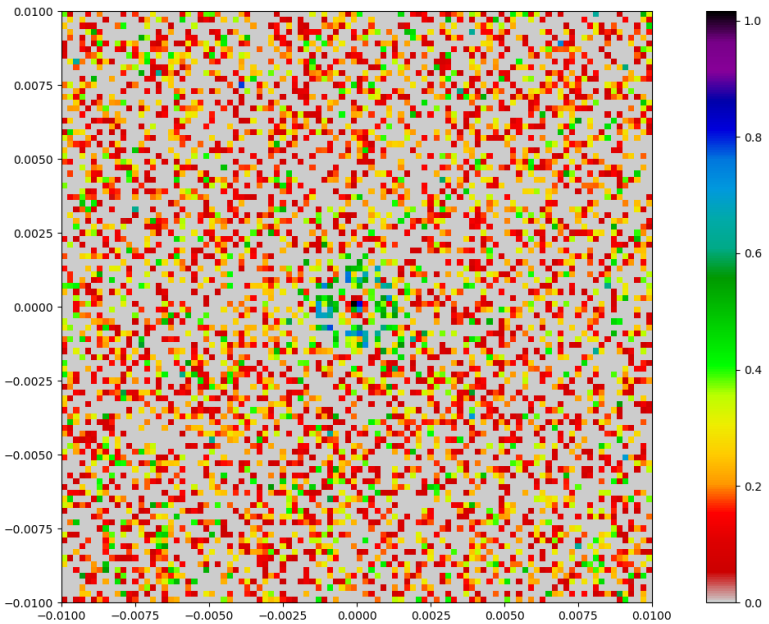

Figure 8: Monte Carlo plot with low resolution and a low number of samples

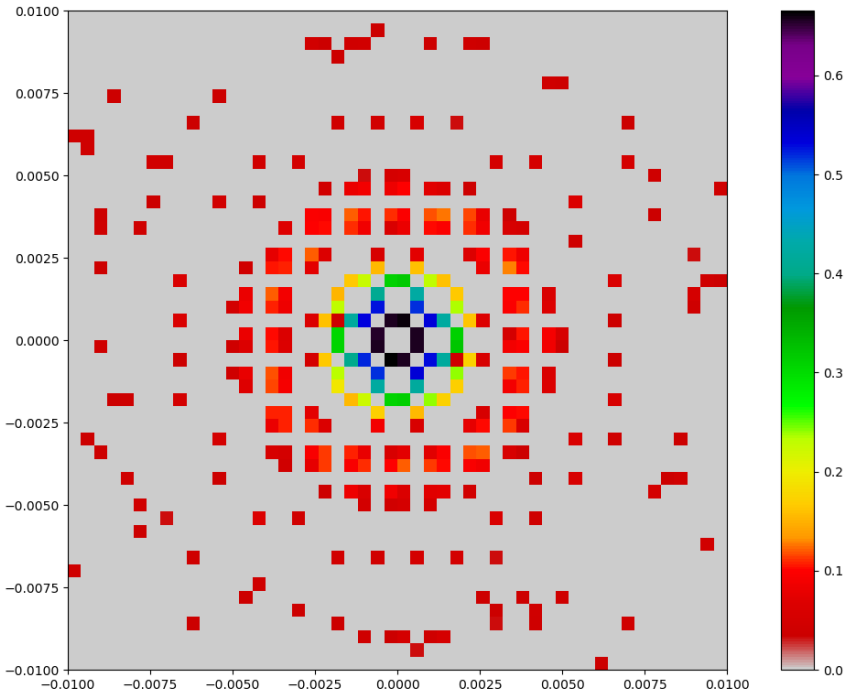

Figure 9: Monte Carlo plot with low resolution and a high number of samples

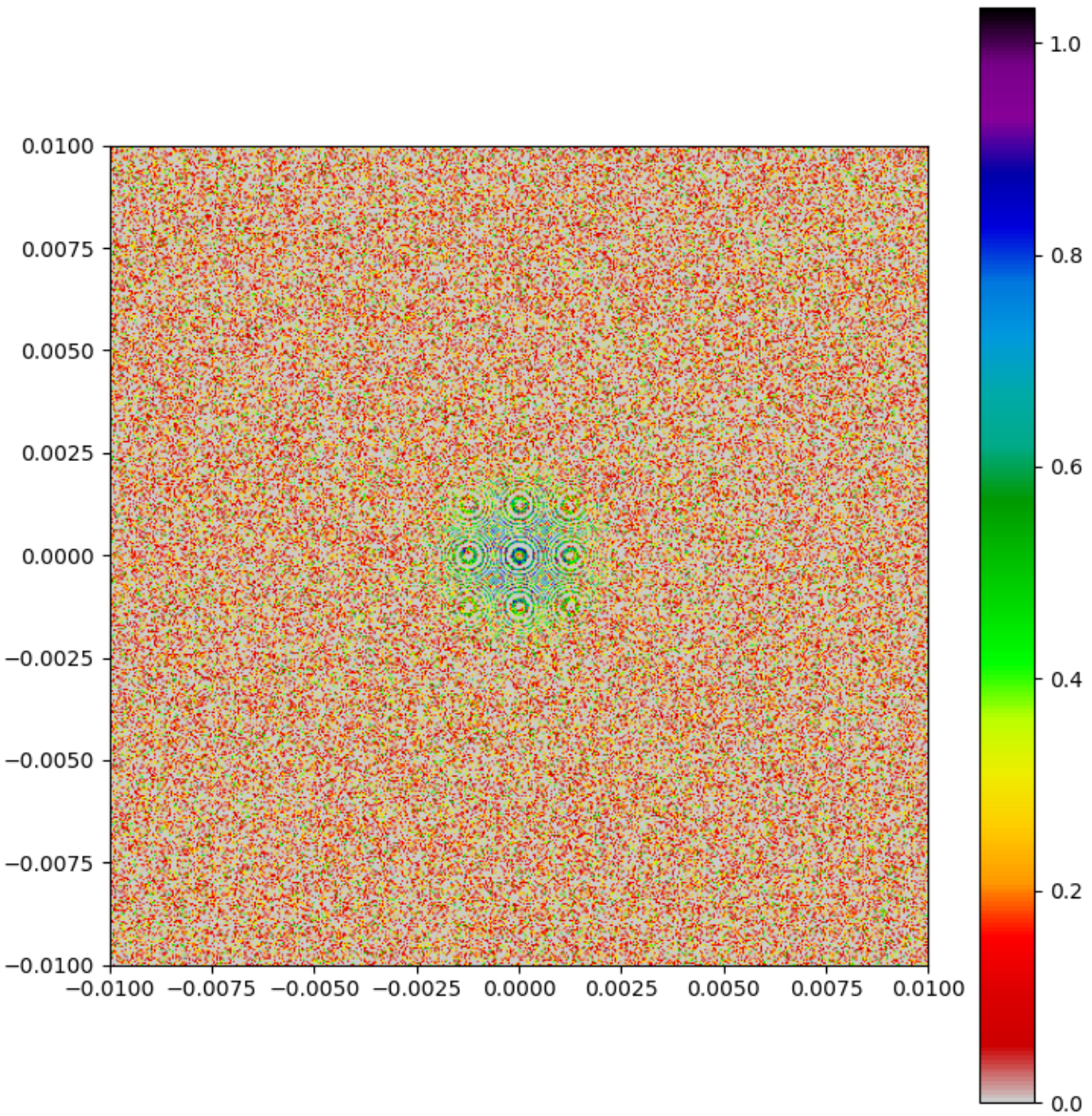

Figure 10: Monte Carlo plot with high resolution and a low number of samples

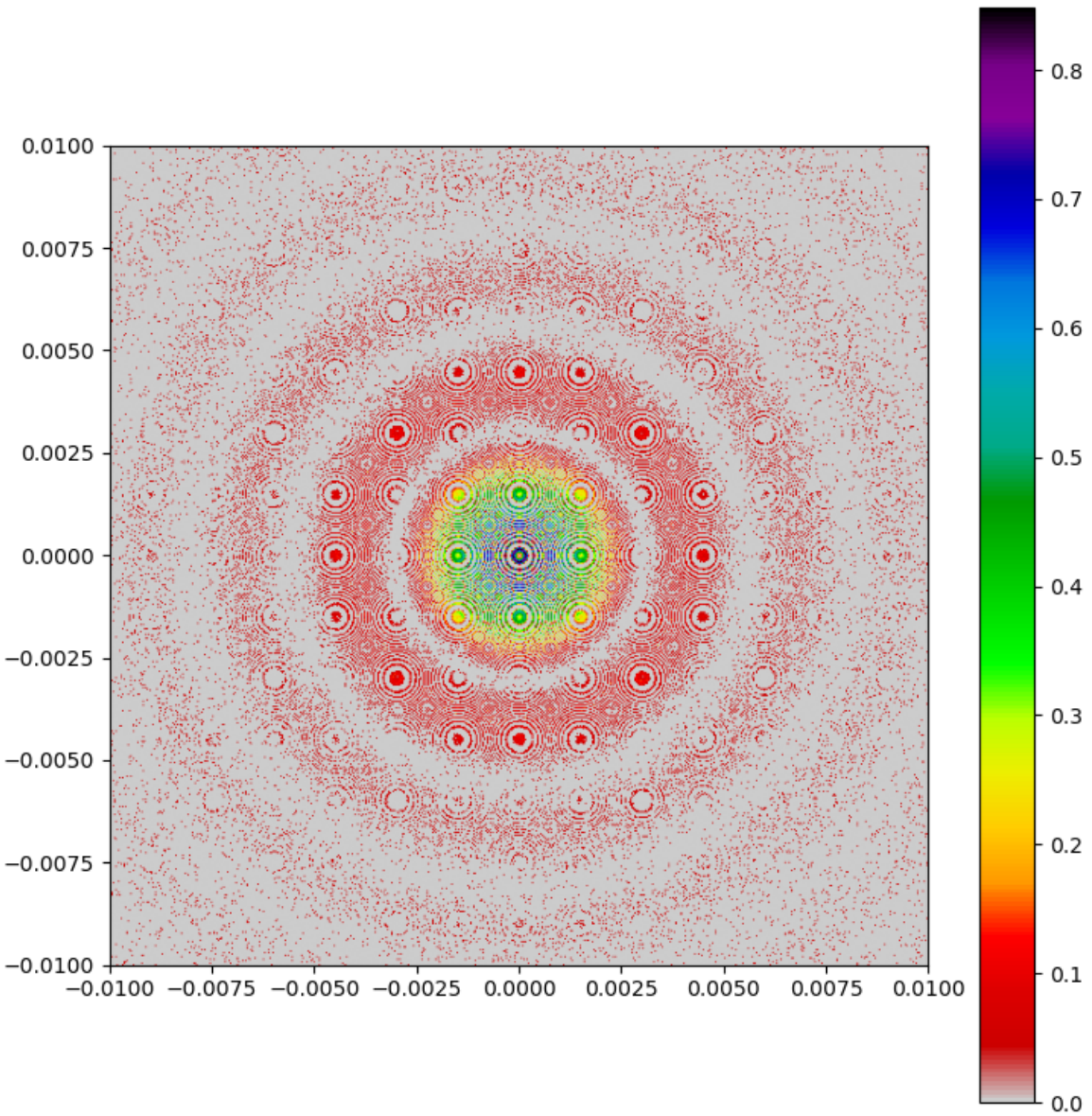

Figure 11: Monte Carlo plot with high resolution and a high number of samples

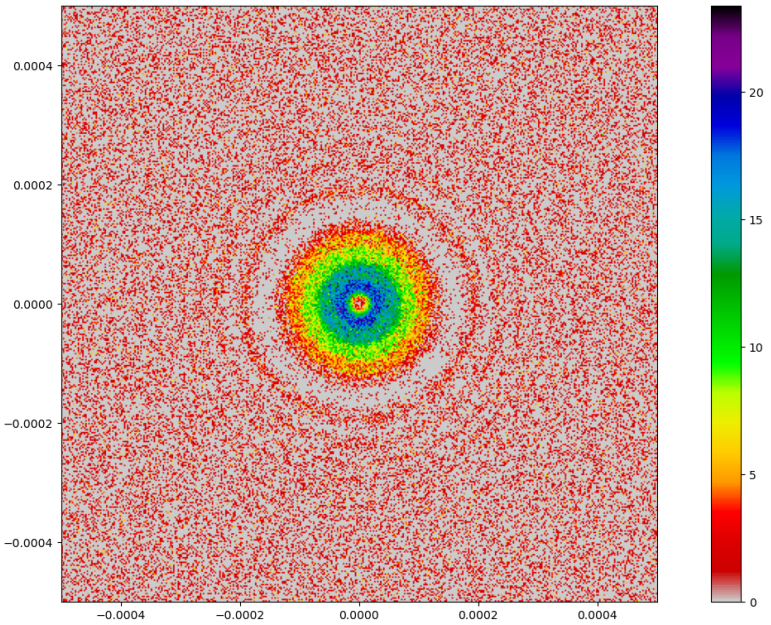

Figure 12: Monte Carlo plot of near field diffraction with high resolution and a high number of samples

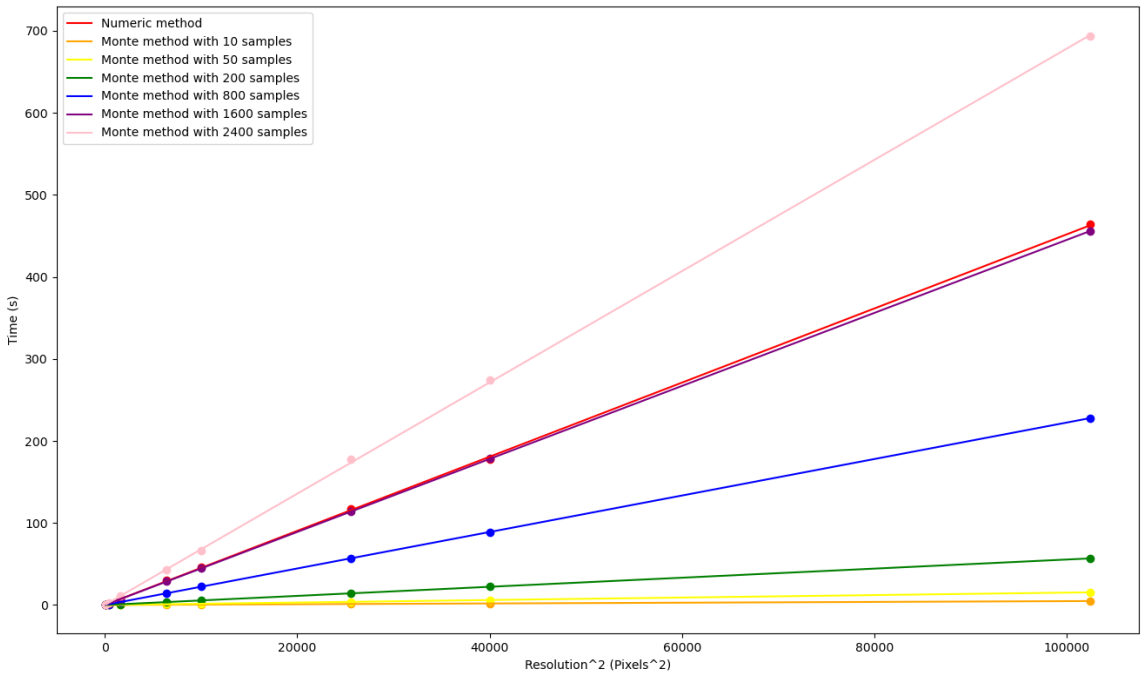

Figure 13: A plot of time against resolution^2 for several different plotting methods

sufficiently large resolution and number of samples, it gave a good plot whilst still being more efficient than the numerical method.