# Fringe Visibility Project

OSC Python Workshops, Fall 2023

## Introduction

Suppose we take a laser and attenuate it enough that only one photon is being emitted at a time. If we use this attenuated laser in a Mach-Zehnder interferometer, would we still expect to see an interference pattern? This phenomenon is called single photon interference, and can be explained with quantum mechanics. In a single photon interference experiment with a Mach-Zehnder interferometer, our goal is to measure the visibility of a fringe pattern at multiple different angles of input polarization. To accomplish this, we will write a Python script capable of taking an image of a fringe pattern and computing its visibility.

We will break down this script into several smaller projects, some more approachable than others, so everyone is encouraged to try different components of this script. The difficulty of each portion of this script will be ranked out of 10, which is based on both the required programming skill and optics background. Each task will be headed using the format "**Task Description (difficulty/10):**."

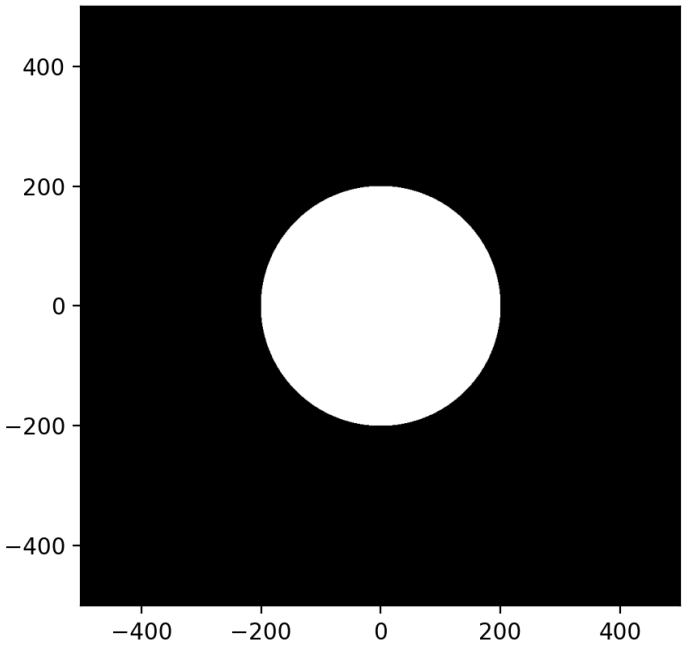
## Beam Detection Simulation (?/10):

### Part (a): Single Beam Simulation

Suppose you have a circular beam of radius $R$ pixels incident on a $N_x \times N_y$ pixel detector. Assume the beam has an intensity of 1. We want to plot the expected output of the detector.

1. Use NumPy to create an $N_x \times N_y$ array where all values are initially set to 0. Then, identify a circle of entries in the array of radius of $R$, and set them equal to 1.

2. Use Matplotlib's "imshow" command to plot this array. You should get an image like the following:




   Try to set the axis ticks such that the circle is centered at (0,0).

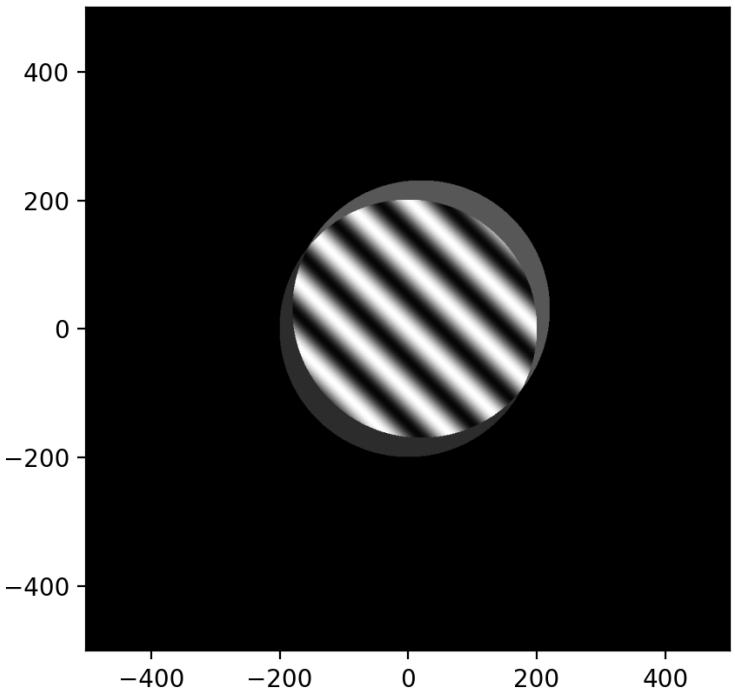
3. **Optional Challenge**: Rather than setting the pixels to either 0 or 1, assign a value between 0 and 1 depending on approximately how much of the pixel lies within the circle. Treat each coordinate as the location of the center of the pixel. If the pixel is fully within the circle, set the value to 1. If the pixel is completely outside the circle, set the value to 0. We recommended that you use an approximation for this task, as the math for an exact evaluation can get quite involved.

### Part (b): Tip-tilt Interference Pattern

Now, we will model two beams which create a tip/tilt interference pattern with one another. Write a method that takes the following inputs:

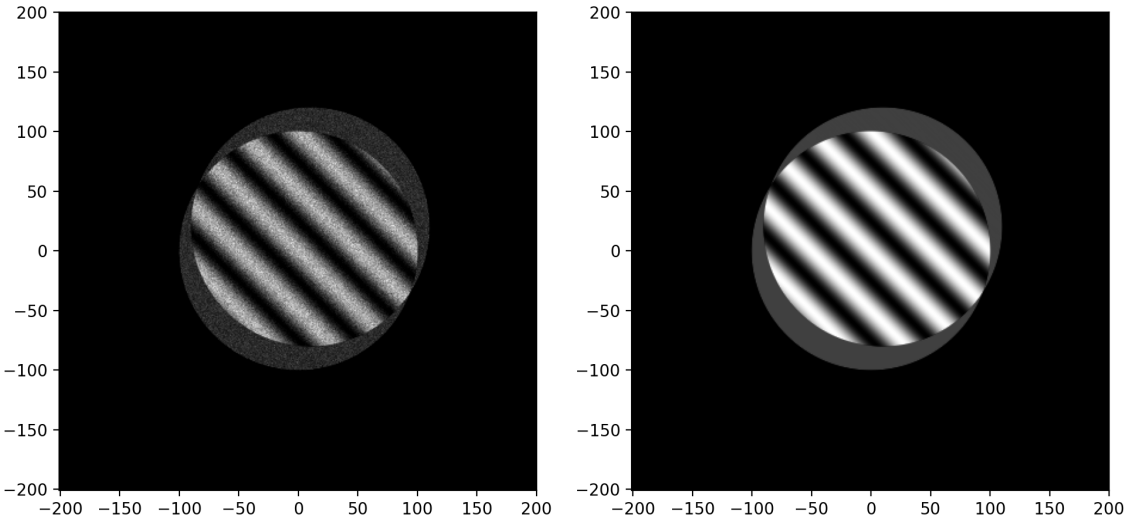
1. $E_1$: Amplitude of the first beam,
2. $E_2$: Amplitude of the second beam,
3. $R$: Radius of the beams,
4. $x_0$: Offset of the beams in the $x$ direction,
5. $y_0$: Offset of the beams in the $y$ direction,
6. $\varphi_x$: Phase of tilt in the $x$ direction,
7. $\varphi_y$: Phase of tip in the $y$ direction,

then computes the image of the interference pattern between these two beams. If you add tip and tilt, offset the beams, and set them to have different amplitudes, you should get an image similar to the following:



## Part (c): Initial Detection Simulation

The data generated in part (b) can be interpreted as the relative probabilities for a photon hitting a given pixel on the detector. Write a method that simulates the image captured after $N$ photons hit the detector. For this task, we recommend that you use the 'bisect' method.

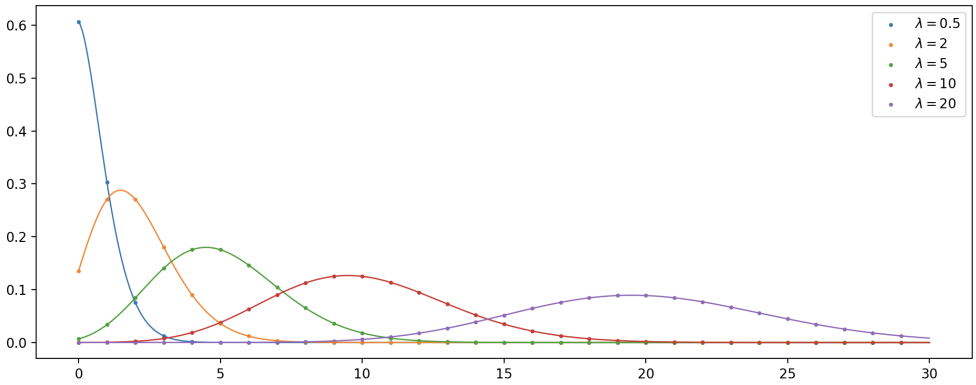



The image on the left is generated from simulating $10^6$ photon detection events using probabilities from the image on the right, where we have a 400 x 400 pixel detector and a 100 pixel radius beam. This simulation can take a lot of time, so you may want to further reduce the size of the detector and the beam in order for the script to run faster.

## Part (d): Noise Modeling

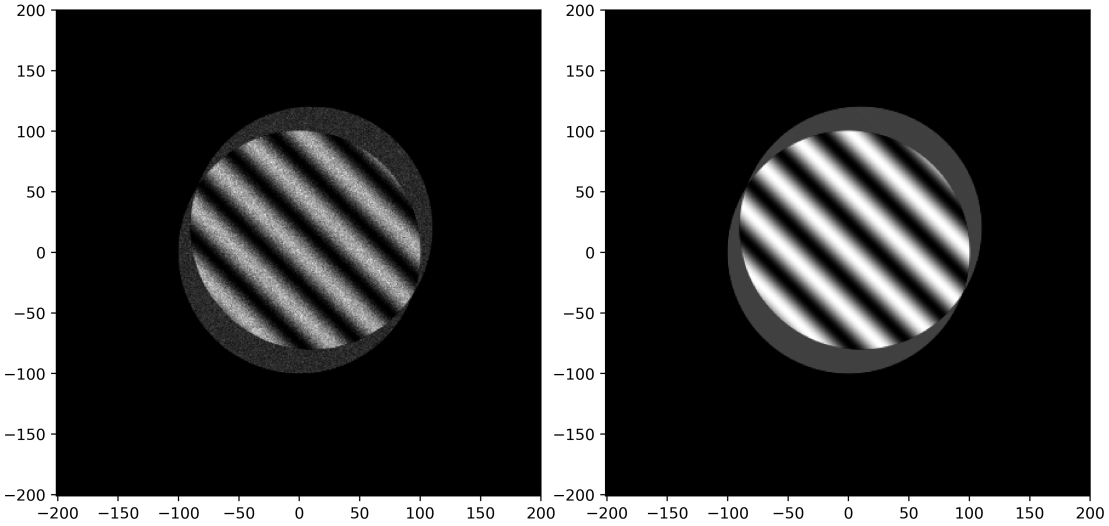
Now, we want to artificially add noise to the image. We will consider the following sources of noise:

1. **Shot noise**: This is the noise which we simulated in part (c), which is a consequence of the random variation in detection events inherent to the discrete nature of the photon signal we are collecting over a finite time interval. Instead of modeling this noise by running a simulation, as done in part (c), shot noise can be modeled with a Poisson distribution. The Poisson distribution has a different shape dependent on the desired mean value, which in this case is the number of expected photon detection events. A plot of the Poisson distribution for various values of expected photons is depicted below, where $\lambda$ denotes the photon expectation value.

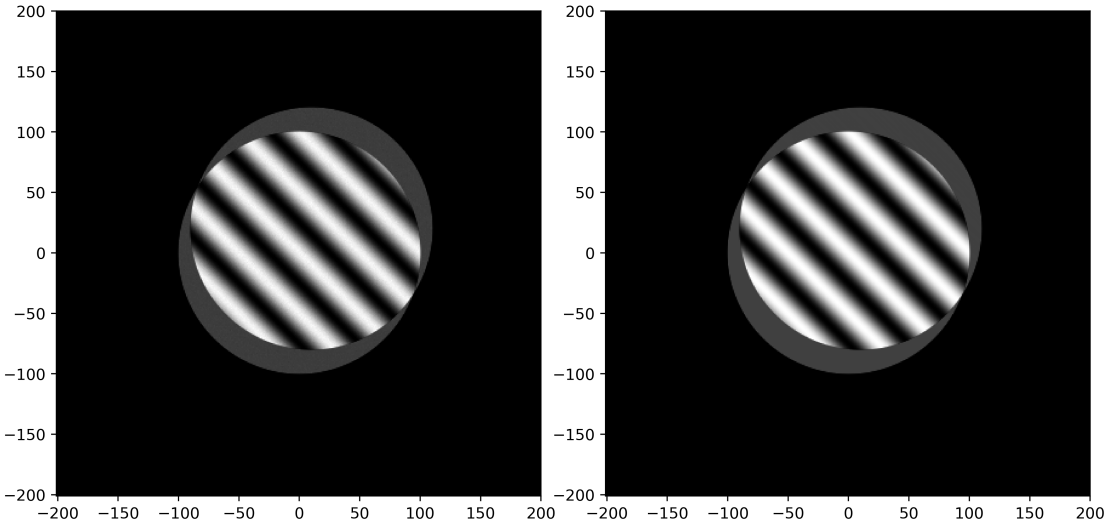



Note that the dots on the plot are the only possible outputs from the distribution, as the Poisson distribution is discrete and solely outputs integers.

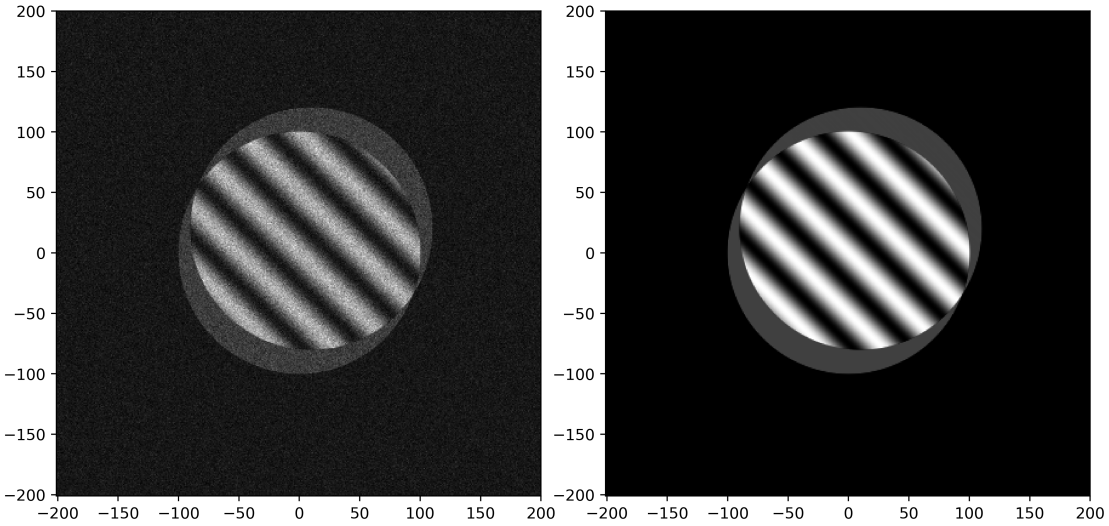
We can artificially model shot noise by individually sampling a Poisson distribution for each pixel based on its expected number of photon counts. The benefit of this approach is that we can generate the simulated images significantly faster. Below is an image generated by artificially modeling shot noise for the same parameters used in part (c). Note the similarities between the image generated in part (c) and the image below.




The signal to noise ratio (SNR) is proportional to $\sqrt{N}$, where $N$ is the number of detection events. This means that the quality of the image will improve as $N$ increases. We repeated the simulation with $10^8$ rather than $10^6$ photons and generated the following image, whose SNR should then be 10x larger than the prior simulation:

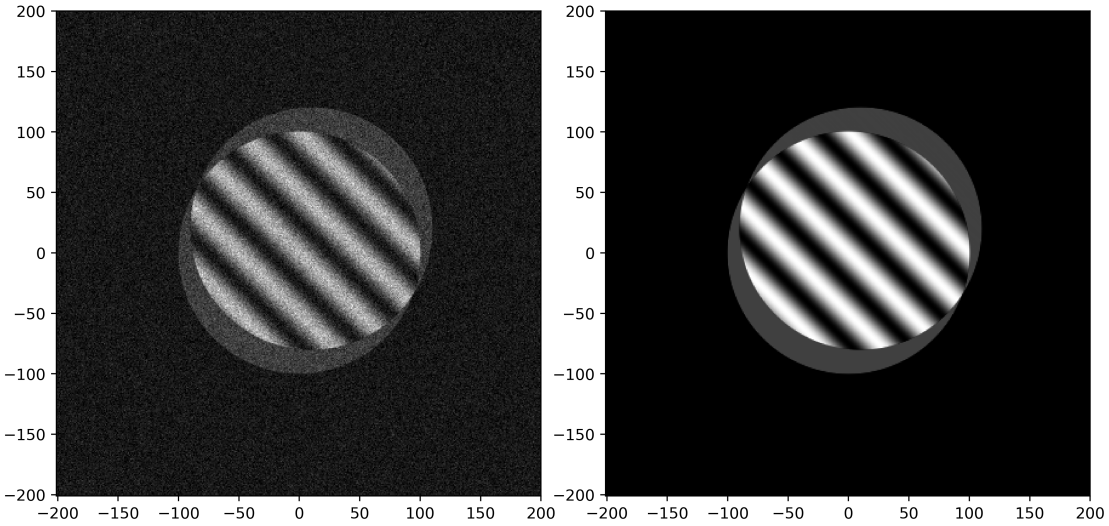



2. **Dark current noise**: Occasionally, pixel detection events are triggered thermally, where energy from the temperature of the detector activates the pixel. This noise is temperature dependent, and detectors are typically cooled to very low temperatures in order to mitigate its effects. Dark current noise is measured in terms of electrons per pixel per second, which we will denote as $d$. For each pixel, the expected number of dark current electrons collected during a given exposure is $d \cdot \tau$, where $\tau$ is the exposure time. Dark current also follows a Poisson distribution, which can then be used to model this noise source. Below is a simulation of the image for $10^6$ photons and $d \cdot \tau = 10$, accounting for both shot noise and dark current noise:




Notice that unlike shot noise, dark current can trigger detection events for pixels which have a 0% probability of being triggered by a photon. Thus, the detector is still registering a "current" even if the pixel is completely "dark." Fortunately, with a properly cooled detector, $d \cdot \tau$ is relatively small.

3. **Read noise**: A detector works by collecting electrons at each pixel during its exposure time, where each photon has some probability of generating an electron, and then these electrons are read out to the camera to generate an image. Read noise occurs during the "reading" step where these electrons are converted to a digital signal sent to the camera. This noise is typically independent of number of electrons collected and the exposure time. For this project, we will model the read noise as a Gaussian distribution with a standard deviation of $\sigma_r$ centered at 0. Note that using a model with negative noise may result in having negative readout values, which should be set to 0. Below is an image which we simulated with all three sources of noise with $10^6$ photons, $d \cdot \tau = 10$, and $\sigma_r = 4e^-$:



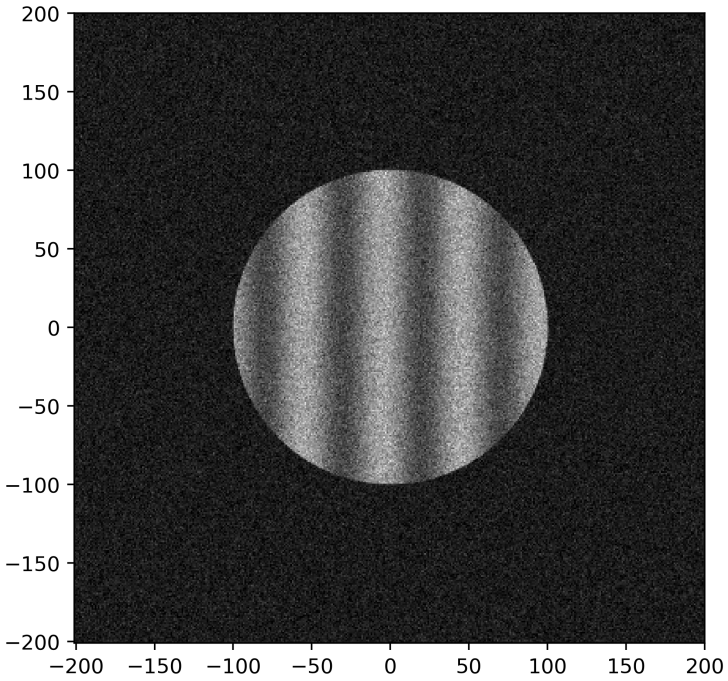

Write a script that takes the following inputs

1. Expected number of photons hitting the detector during the exposure time,
2. Expected number of dark current electrons $d \cdot \tau$ per pixel,
3. Standard deviation $\sigma_r$ of the read noise,

and then simulates a noisy image collected by the detector.

## Visibility Calculations (?/10):

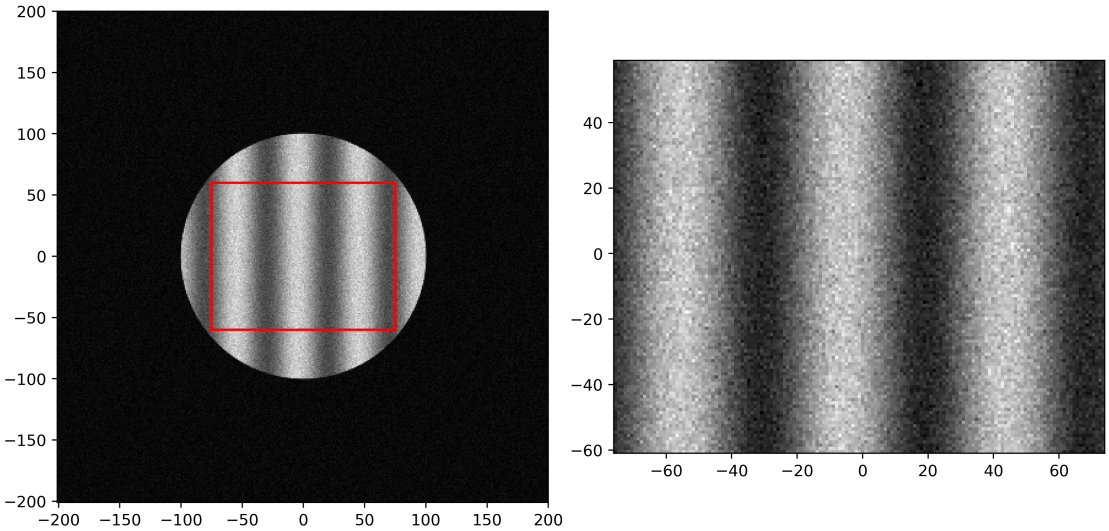
We will now assume that the beams perfectly overlap and the interference is purely tilt, as depicted in the image below. This image is located in the repository with the filename "fringe_simulation.png."



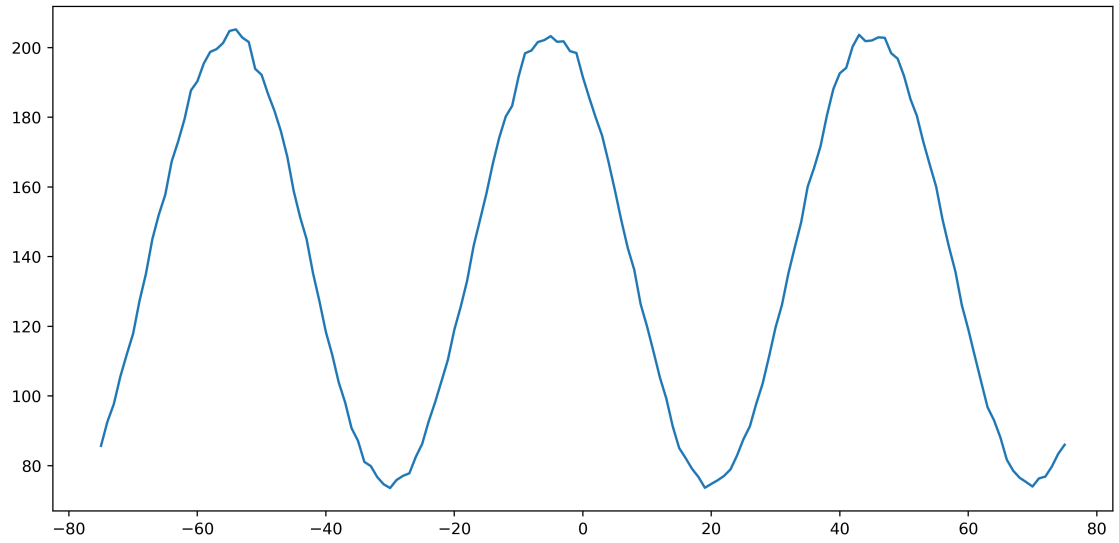

This image was generated with $4 \cdot 10^6$ photons, $E_1 = \cos(15°)$, and $E_2 = \sin(15°)$. Note that the expected visibility of this fringe pattern is 0.5. In this section, we will experiment with a few different methods for computing the fringe visibility of this image.

### Part (a): Averaging Along the y-axis

The first approach we will consider involves selecting a rectangular region within the circle and averaging values along the $y$ axis. Below is an example of such a rectangular region:



If we evaluate the average value of the intensity along the y direction, we get the following plot:




Isolate a rectangular region which is completely within the circle, and then average along the $y$ direction to generate a plot similar to the one above.

### Part (b)

There are two methods we can use to evaluate the fringe visibility using the above data

1. Take $I_{max}$ and $I_{min}$ to be the max and min values of the data, respectively.
2. Fit the data to a sinusoidal function of the form

$$f(x) = C + A\sin(\phi + x\omega),$$

then set $I_{max} = C + A$ and $I_{min} = C - A$.

Implement both of these methods and make sure they result in a value close to 0.5.

### Part (c)

Fourier transforms

## Error analysis (?/10)

In this section, we will use the code developed in the previous sections to repeatedly generate images and evaluate their visibility using each of the methods. Our goal is to evaluate which of the methods has the best performance.

### Part (a)

Write a script that takes the following inputs

1. $N$: Number of iterations,
2. $\theta$: Angle of the polarization filter,
3. Any noise related coefficients used to generate the images,

then generates $N$ images, evaluates their visibility using each method from the previous section, and stores the results in separate arrays depending on the method used. Take averages of each of these arrays, and compare them to the expected value of the visibility for a given $\theta$. Recall

$$E_1 = E_0 \cos(\theta), \qquad E_2 = E_0 \sin(\theta), \qquad v(\theta) = |\sin(2\theta)|.$$

### Part (b)

Generate histograms of the visibility results for each of the three methods using the data you've collected. Which method appears to work the best, and why? Repeat this a few times with different levels of noise to analyse how noise impacts the performance of each method. You may find that the best method depends on the noise levels, but this will depend on your implementations.