
I started using Julia for this homework assignment as I've been fighting small Python syntax and just feel more comfortable with Julia at this point. Tom said it was okay by him!

Problem 1 (Frequency modulation).

Solution.



Figure 1: Buster Keaton

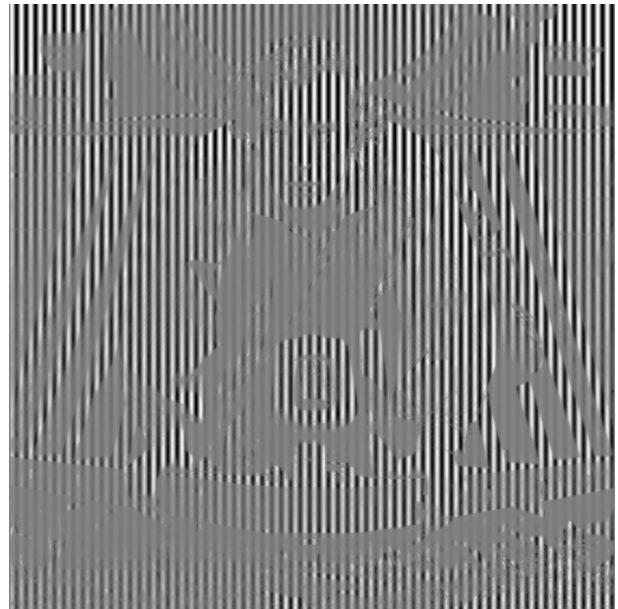


Figure 2: ...after applying a sine wave.

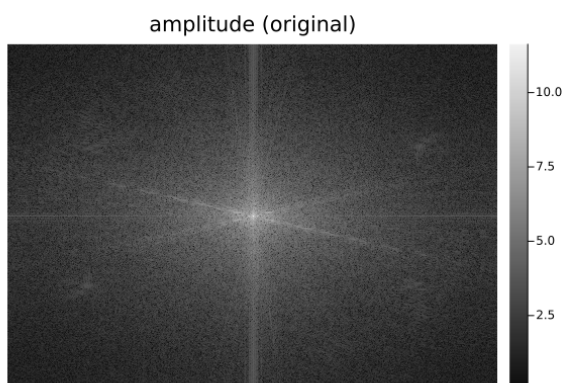


Figure 3: Amplitude array of original image after a Fourier transform.

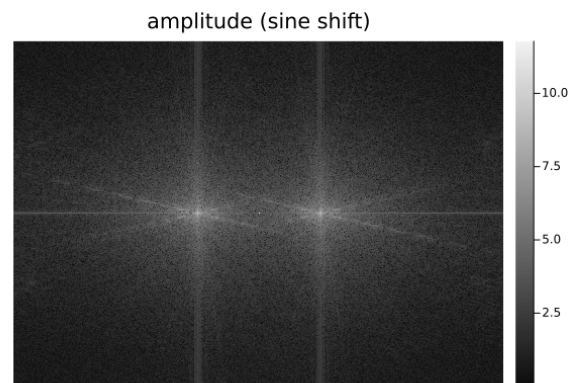


Figure 4: Amplitude array of sine modded image after a Fourier transform.

The amplitude array of the original image is shown in Figure 3 whereas the amplitude array of the image after a sine modification is shown in Figure 4. As expected, we can see a frequency array in the original image with a central peak. In the sine modulated image, we see two peaks instead. These two peaks (I believe) correspond to the \pm values of a sine wave. We lose some intensity of the peak at the center because most of the image is washed out by the sine mod, but we can still see faint traces of it.

Problem 2 (Quantized aggregates).

Solution. Alright, so my approach goes like this: I first made sure I could at least extract the emitters with the non-noise image. I wrote a function that iterates through multiples of 33px in the x and y directions and pulled just those pixels, given that the problem states we can take that as a truth. I verified this worked because the resulting array had length 100. I looked at these values and saw that they had some distinct gaps in their ‘typical’ value. Plotting them in a histogram gave me the results seen in Figure 5. This gave me distinct peaks to look at for classifying monomers, dimers, and trimers. I then wrote a function sorting these and counting them, which gave me the correct result of 69, 20, and 11 monomers, dimers, and trimers respectively. The trouble arose when I tried this with the noisy image, where the histogram can be seen in Figure 6.

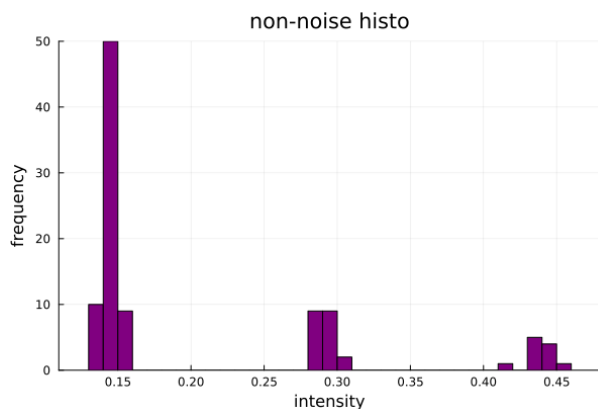


Figure 5: Emitter values in the no-noise image.

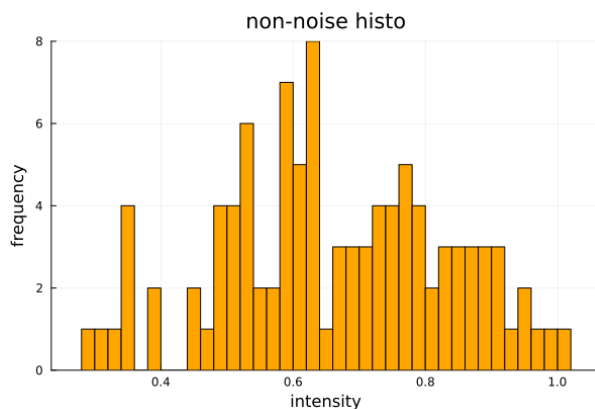


Figure 6: Emitter values in the image that contained a gradient noise overlay.

At this point I needed to find a way to subtract the noise. My process was to blur out the background, emphasizing the emitter pixels, then subtract that blurred background from the original image. From there, I would try to extract the emitters using the functions I wrote before. I then looked at a sorted list of those values and found certain ‘jumps’ in the values, which is where I set the values for counting and arrived at what I believe is the solution: 25 monomers, 11 dimers, and 64 trimers. The main two functions used are in Figures 9 and 10, and the function used for the Gaussian blur is a part of the `Images` package in Julia.

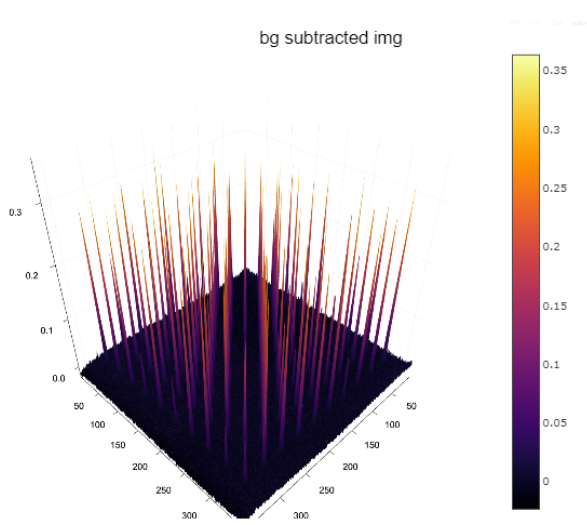


Figure 7: Noisy emitters after subtracting background (iso view).

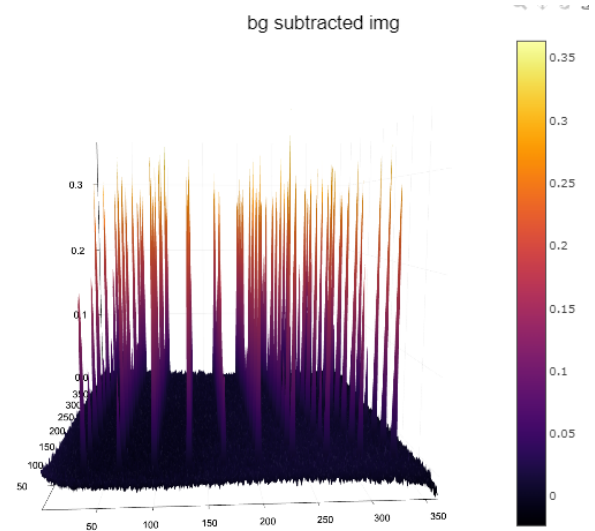


Figure 8: Noisy emitters after subtracting background (side view).

```
function extract_emit(img)
    vec = []

    for i = 33:33:size(img)[1]
        for j = 33:33:size(img)[2]
            push!(vec, img[i, j])
        end
    end

    return vec
end; ✓
```

Figure 9: Function for extracting only the emitter pixels.

```
function count_emitters_noise(vec, a, b, c)
    mono, dim, tri = [], [], []

    for i = 1:length(vec)
        if vec[i] ≤ a
            push!(mono, vec[i])
        elseif a < vec[i] ≤ b
            push!(dim, vec[i])
        elseif b < vec[i] ≤ c
            push!(tri, vec[i])
        end
    end

    return mono, dim, tri
end; ✓
```

Figure 10: Function for counting the emitters within certain values specified by a , b , and c .

Problem 3 (A high-resolution PSF).

Solution. This problem was fairly straightforward as everything is provided for us so I'm going to start by posting my function so the comments can be seen to show what's going on. I also love that Julia lets me use unicode for variables (hence the green and blue by λ). Below this you'll see the results of the function I wrote.

```
# Functions
# euclidean distance calculator
function euclid_dist(cent, scale, a, b)
    A = (a - cent) * scale
    B = (b - cent) * scale
    C = sqrt(A^2 + B^2)

    return C
end;

function psf(N::Int, λ::Float64, aper::Float64, scale::Float64)
    # Make a grid of distances from the center of an NxN matrix
    center = (N + 1) / 2 # calc the center
    r = zeros(N, N) # initiate a matrix of distances from center

    # for distances from the center, calculated the euclidean distance
    for i = 1:1:N
        for j = 1:1:N
            r[i, j] = euclid_dist(center, scale, i, j)
        end
    end

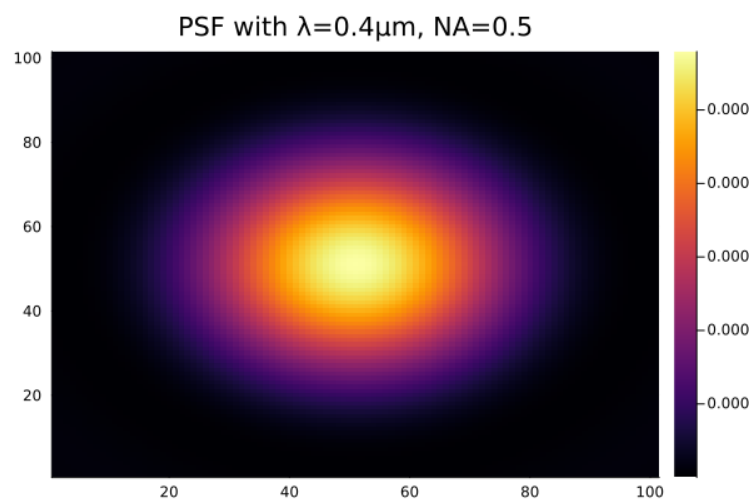
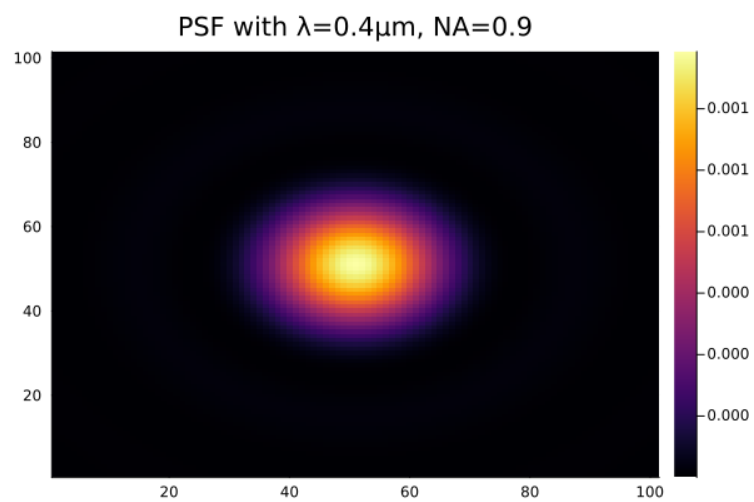
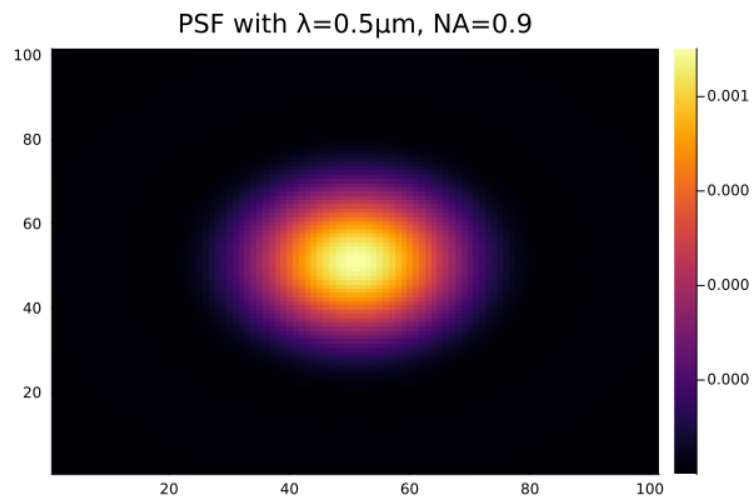
    # now multiply the matrix r by some scalar values
    v = (2 * π / λ) * aper * r

    # initiate an array for PSF values
    psf_array = zeros(N, N)

    # Now do PSF calc for each point in psf_array
    for i = 1:1:N, j = 1:1:N
        if v[i, j] == 0
            psf_array[i, j] = 1 # special case for v = 0
        else
            # In the SpecialFunctions package, we get bessell functions
            psf_array[i, j] = 4 * (besselj1(v[i, j]) / v[i, j])^2 # given point spread function
        end
    end

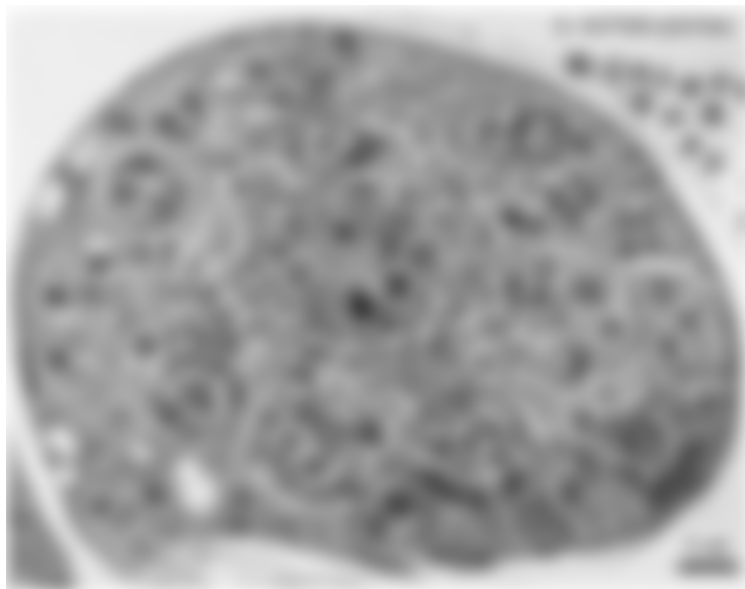
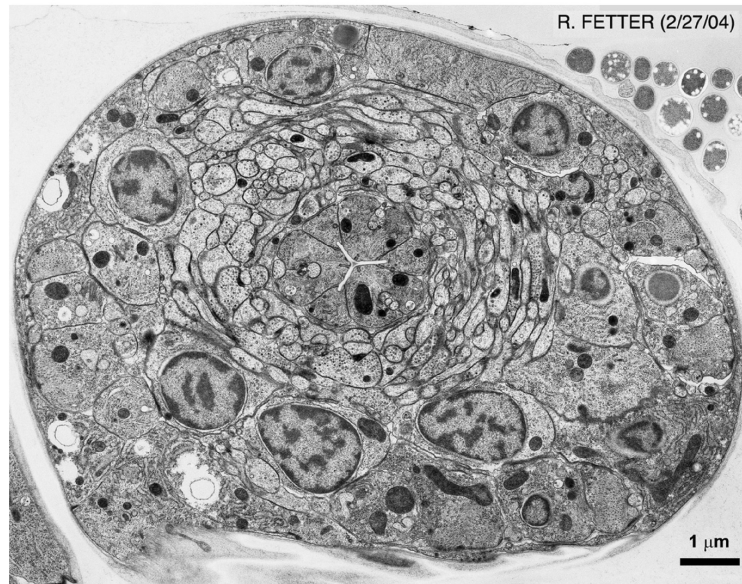
    return psf_array
end;

# variables to tweak
N = 101;
λ = 0.5;
λ = 0.4;
aper1 = 0.9;
aper2 = 0.5;
scale = 0.01;
```



Problem 4 (A worse worm image).

Solution. I'm a little confused on this one - would the image just be blurred? My only thought, since it hints at using the PSF from before, would be to apply that as a filter to the image. This makes sense to me conceptually since the PSF is determined by apertures and wavelength so if we used the indicated values and apply that as a filter to the image, it would simulate it as if we're looking at visible light using a specific aperture. The figures below show the original image and what it would look like when using the PSF as a filter. I chose the scale to be 1/100 as that's what we chose for similar scales in the previous problem.



Problem 5 (SNR and Poisson noise).

Solution. I read that one definition of the signal to noise ratio is the mean divided by the standard deviation. For a Poisson distribution, the mean would be equal to N_{photon} or the number of photons expected. The standard deviation would then be $\sqrt{N_{photon}}$. Therefore

$$SNR = \frac{\mu}{\sigma} = \frac{N_{photon}}{\sqrt{N_{photon}}} = \sqrt{N_{photon}}. \quad (1)$$

Problem 6 (Simulated point-sources, part 1).

Solution. We can see that when we add noise, for 50 photons we don't catch much data as far as localization goes. However, for 500 photons, we see more definite positions for the actual position. I checked by making this huge and saw that it essentially just becomes a centered psf again, as expected.

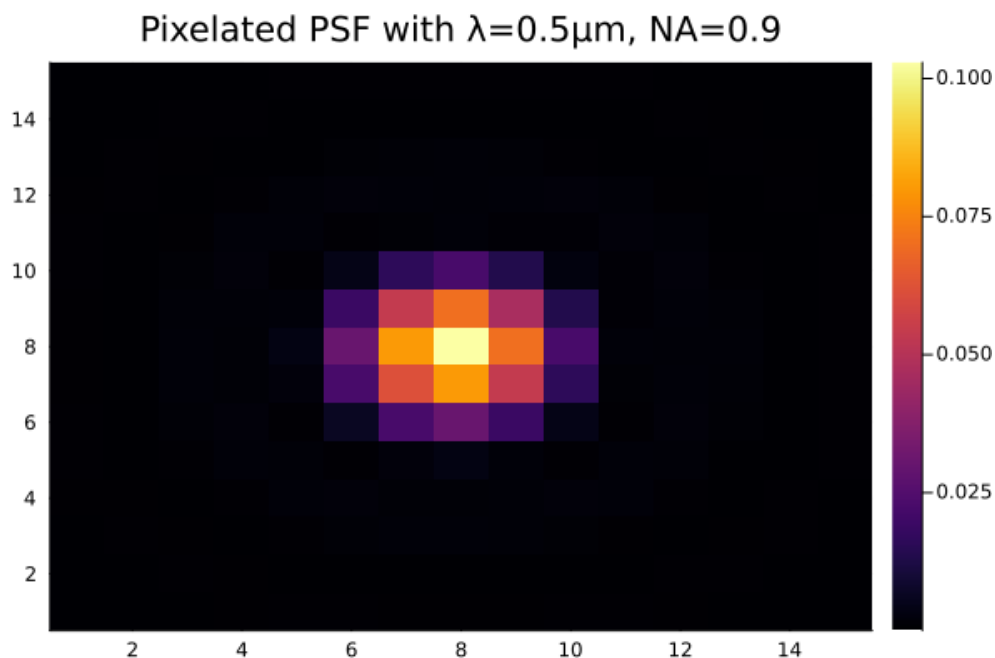


Figure 11: Pixelized psf function that adds pixelation to the source.

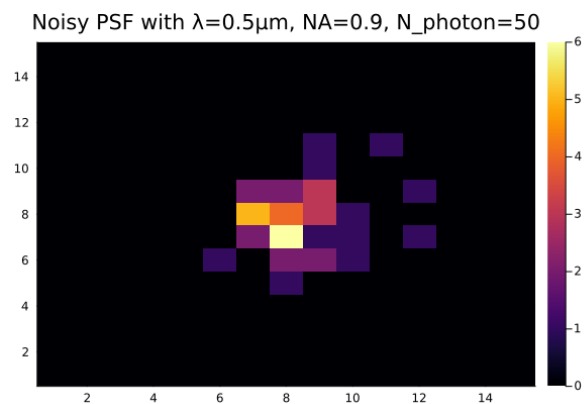


Figure 12: A psf with Poisson noise added and photon number 50.

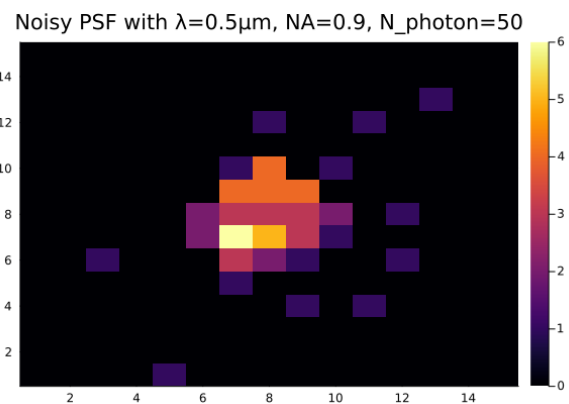


Figure 13: Another psf with Poisson noise added and photon number 50.

Noisy PSF with $\lambda=0.5\mu\text{m}$, $\text{NA}=0.9$, $N_{\text{photon}}=500$

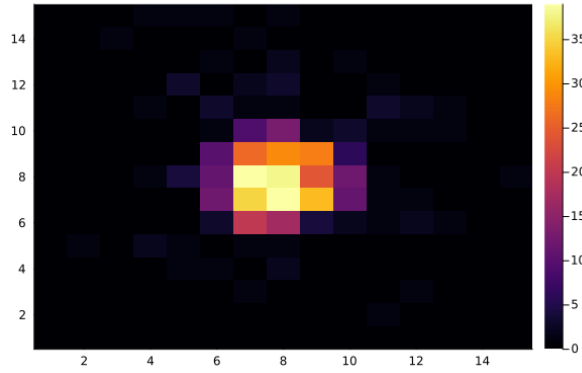


Figure 14: A psf with Poisson noise added and photon number 500.

Noisy PSF with $\lambda=0.5\mu\text{m}$, $\text{NA}=0.9$, $N_{\text{photon}}=500$

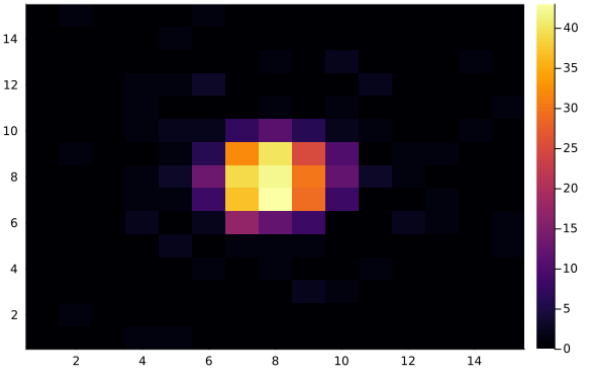


Figure 15: Another psf with Poisson noise added and photon number 500.

```
function psf_pixelization_noisy(N::Int,
    λ::Float64,
    aper::Float64,
    cam_scale::Float64,
    fine_scale::Float64,
    Np::Int)

    # need to calculate the psf for a fine grid
    fine_N = Int(N * cam_scale / fine_scale)
    fine_psf = psf(fine_N, λ, aper, fine_scale)

    # down sample that feller
    coarse_psf = zeros(N, N) # initiate
    block_size = Int(cam_scale / fine_scale) # define the block size

    for i=1:1:N, j=1:1:N # iterate through using block sizes
        fine_i = ((i - 1) * block_size + 1):(i * block_size)
        fine_j = ((j - 1) * block_size + 1):(j * block_size)
        coarse_psf[i, j] = sum(fine_psf[fine_i, fine_j])
    end

    # scale and add noise for part (b)
    coarse_psf_scaled = coarse_psf * (Np / sum(coarse_psf))

    noisy_psf = rand.(Poisson.(coarse_psf_scaled))

    return noisy_psf # return the normalized version
end;
```

Figure 16: Code for generating a noisy PSF.

Problem 7 (Simulated point-sources, part 2).

Solution.

```
# Pixelization and noise
function psf_pixelization_noisy(N::Int,
    λ::Float64,
    aper::Float64,
    cam_scale::Float64,
    fine_scale::Float64,
    Np::Int,
    xc::Float64=0.0,
    yc::Float64=0.0)

    # need to calculate the psf for a fine grid
    fine_N = Int(N * cam_scale / fine_scale)
    fine_psf = psf(fine_N, λ, aper, fine_scale, xc, yc)

    # down sample that feller
    coarse_psf = zeros(N, N) # initiate
    block_size = Int(cam_scale / fine_scale) # define the block size

    for i=1:1:N, j=1:1:N # iterate through using block sizes
        fine_i = ((i - 1) * block_size + 1):(i * block_size)
        fine_j = ((j - 1) * block_size + 1):(j * block_size)
        coarse_psf[i, j] = sum(fine_psf[fine_i, fine_j])
    end

    # scale and add noise for part (b)
    coarse_psf_scaled = coarse_psf * (Np / sum(coarse_psf))

    noisy_psf = rand.(Poisson.(coarse_psf_scaled))

    return noisy_psf # return the normalized version
end;
```

Figure 17: Pixelized psf function but with the offsets xc, and xy. These are implemented directly to the psf function from before, which actually passes them to the euclidean distance function also from before.

```

# Functions, first two are from last homework
# euclidean distance calculator
function euclid_dist(cent, scale, a, b, xc, yc)
    A = (a - cent) * scale - xc
    B = (b - cent) * scale - yc
    C = sqrt(A^2 + B^2)

    return C
end;

function psf(N::Int, λ::Float64, aper::Float64, scale::Float64, xc, yc)
    # Make a grid of distances from the center of an NxN matrix
    center = (N - 1) / 2 # calc the center
    r = zeros(N, N) # initiate a matrix of distances from center

    # for distances from the center, calculated the euclidean distance
    for i = 1:1:N
        for j = 1:1:N
            r[i, j] = euclid_dist(center, scale, i, j, xc, yc)
        end
    end

    # now multiply the matrix r by some scalar values
    v = (2 * π / λ) * aper * r

    # initiate an array for PSF values
    psf_array = zeros(N, N)

    # Now do PSF calc for each point in psf_array
    for i = 1:1:N, j = 1:1:N
        if v[i, j] == 0
            psf_array[i, j] = 1 # special case for v = 0
        else
            # In the SpecialFunctions package, we get bessell functions
            psf_array[i, j] = 4 * (besselj1(v[i, j]) / v[i, j])^2 # given point spread function
        end
    end

    return psf_array / sum(psf_array)
end;

```

Figure 18: The xc and yc offset are passed through the functions but actually implemented in my euclid.dist function.

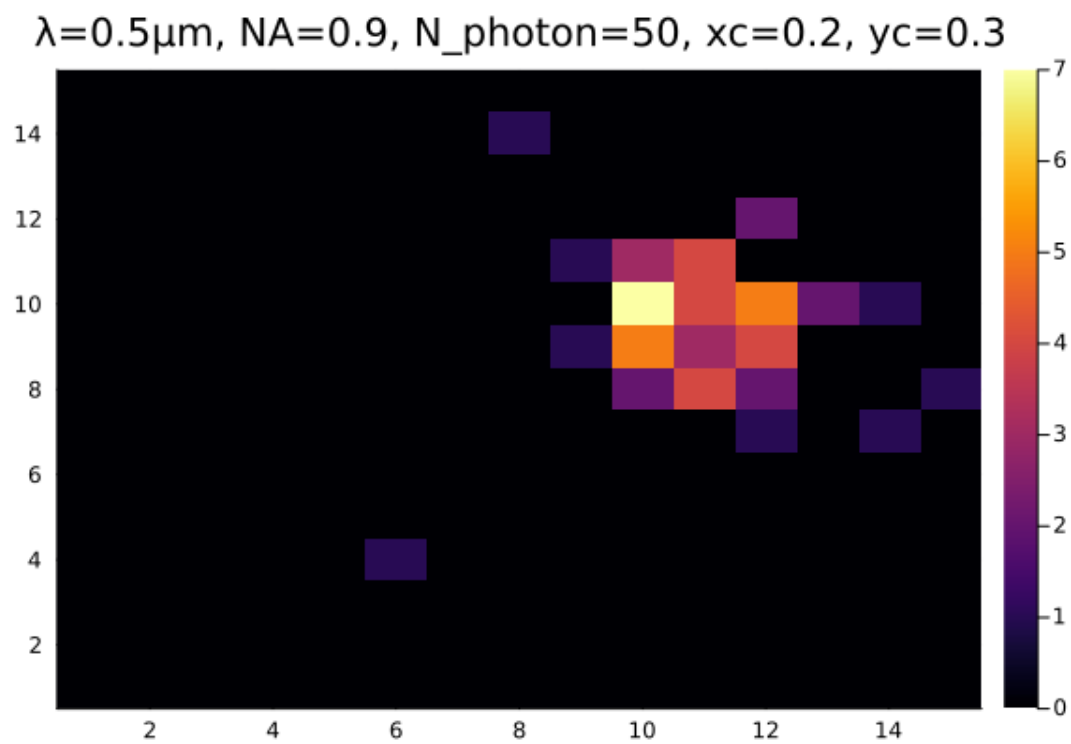


Figure 19: A signal offset by some values in x and y .

Problem 8 (Simulated point-sources, part 3).

Solution.

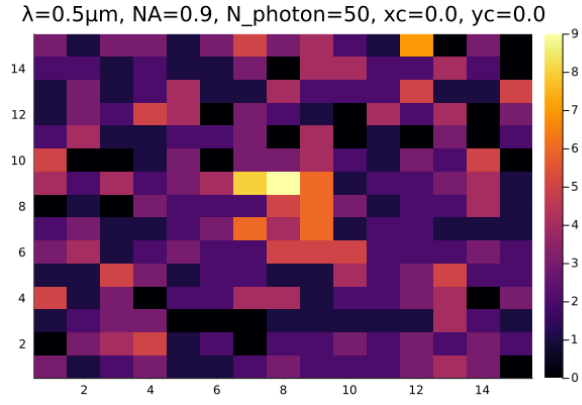


Figure 20: Added background noise with no offset.

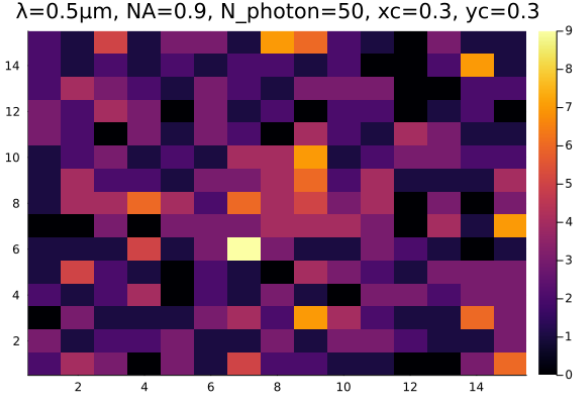


Figure 21: Added background noise with offset $0.03\mu m$ in both directions.

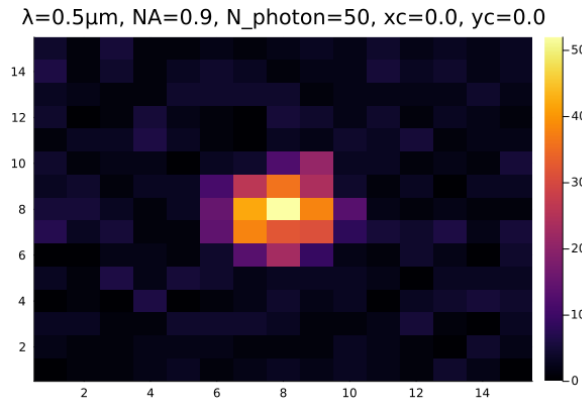


Figure 22: Added background noise with no offset, $N_{\text{photon}} = 500$.

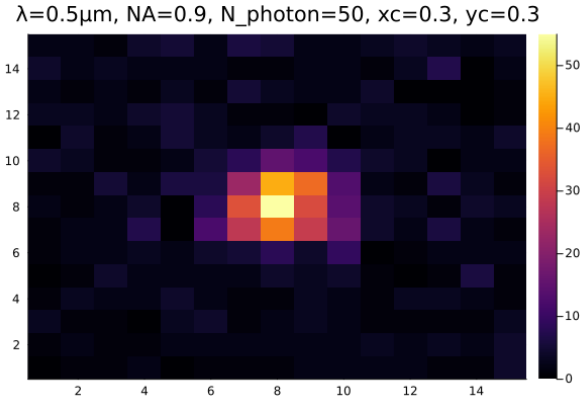


Figure 23: Added background noise with offset $0.03\mu m$ in both directions, $N_{\text{photon}} = 500$.

```
# Add background noise
bg_noise = rand.(Poisson(bg), N, N)
noisy_psf = noisy_psf .+ bg_noise
```

Figure 24: Change to psf function for bg noise.

Problem 9 (Simulating a ring).

Solution. To simulate a ring, I would iterate through the process of making a psf for a point source, but alter the code to consider some distance from the center at a fixed radius. We could alter the offset version of the euclidean distance (or even just add a new function that calls this) to make sure that r is fixed but add some angle offset. Then iterate from $0 \rightarrow 2\pi$ at different intervals to simulate a ring or a ring of lights (if altering the intervals to be large).