HW4_AD

October 23, 2024

[156]:

```
import numpy as np
from skimage import io, filters
from scipy.ndimage import convolve
from skimage.filters import gaussian
import matplotlib.pyplot as plt
```

[152]:

```
def showim(im_array, figsize = (11,5), show_hist = False, nbins = None, cmap =␣
↪'gray', vmin = None, vmax = None):

    if show_hist:
        im_flattened = im_array.ravel()
        if nbins == None:
            number_of_bins = (np.ceil(np.max(im_flattened)) - np.floor(np.
↪min(im_flattened))).astype(np.int64)
        else: nbins = nbins

        plt.figure(figsize=figsize)
        plt.subplot(1,2,1)
        plt.imshow(im_array, cmap=cmap, vmin=vmin, vmax=vmax)
        plt.axis('off')

        plt.subplot(1,2,2)
        plt.hist(im_flattened, bins=number_of_bins, color='black')
        plt.xlabel('Intensity Value')
        plt.ylabel('Frequency')
        plt.title('Image Intensity Histogram')

        plt.tight_layout()

    else:
        plt.figure(figsize=figsize)
        plt.imshow(im_array, cmap=cmap, vmin=vmin, vmax=vmax)
        plt.axis('off')

def thresh_otsu(im_array, show_results = False):

    if show_results:
```

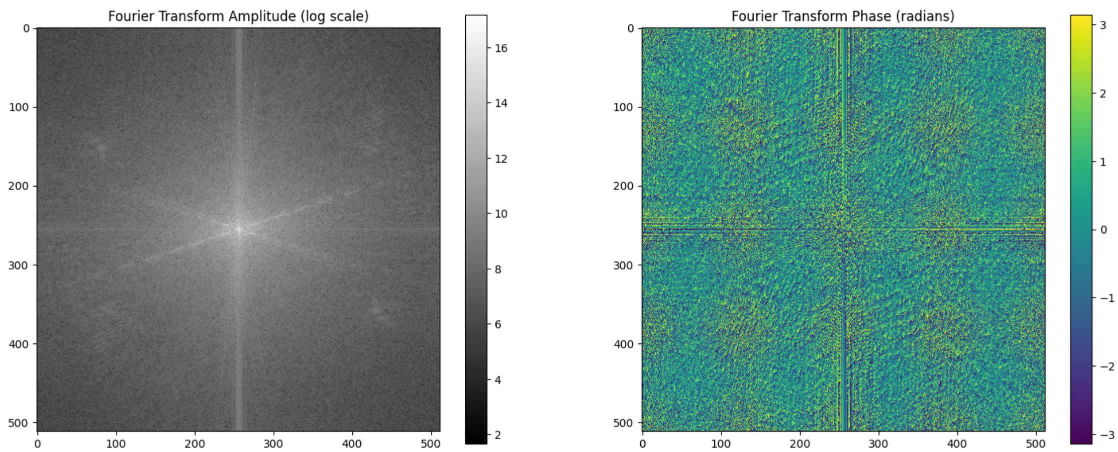
```python
        plt.figure(figsize=(15,7))

        otsu_thresh = filters.threshold_otsu(im_array)
        image_otsu_thresh = (im_array > otsu_thresh).astype(np.uint8)
        plt.subplot(1,2,1)
        plt.imshow(image_otsu_thresh, cmap='gray')
        plt.axis('off')

        im_array_flattened = im_array.ravel()

        plt.subplot(1,2,2)
        plt.hist(im_array_flattened, bins=256, color='black')
        plt.axvline(otsu_thresh, color='red', linestyle='dashed', linewidth=2)
        plt.title('Image Intensity Histogram')
        plt.xlabel('Pixel Intensity')
        plt.ylabel('Frequency')

    else:
        otsu_thresh = filters.threshold_otsu(im_array)
        image_otsu_thresh = (im_array > otsu_thresh).astype(np.uint8)

    return image_otsu_thresh

def surfplot(imarr, figsize = (11,7), cmap = 'viridis', show = True):

    x = np.arange(imarr.shape[1])
    y = np.arange(imarr.shape[0])
    X , Y = np.meshgrid(x, y)

    if show == True:

        fig = plt.figure(figsize=figsize)
        ax = fig.add_subplot(111, projection='3d')
        ax.plot_surface(X, Y, imarr, cmap=cmap)
        ax.set_xlabel('X')
        ax.set_ylabel('Y')
        ax.set_zlabel('Pixel Intensity')
        plt.show();


    return X , Y
```

1) Frequency modulation

Download "Buster_Keaton_General_Train_512.png" from Canvas, a still from Buster Keaton's wonderful film "The General" (1926), which you all should watch, especially since it was filmed in Oregon! I'm sure it's on Kanopy. Also download

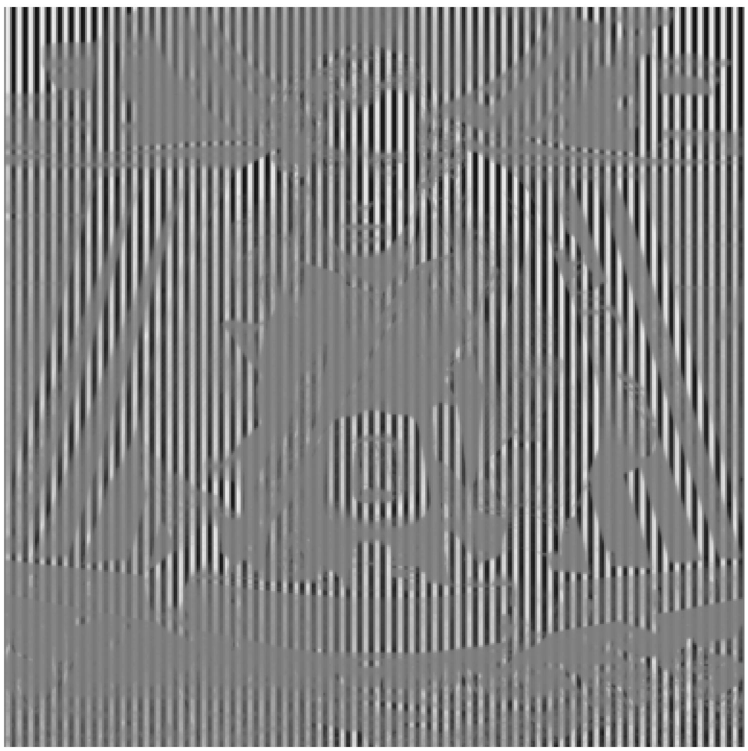
“Buster_Keaton_General_Train_512_sineMod.png”, which is the same image multiplied by a sine wave (sin(2.0 x / Px), with period Px = 8 pixels, scaled to [0, 255]). Fourier Transform each image and show the amplitude arrays. Take the log of the amplitude so that we can see its range easily. You can copy the Fourier Transform code I provided for last week’s homework. Look carefully and describe what’s different between the two Fourier Transforms. Explain why they look like they do; this doesn’t have to be a mathematical proof, but it should be convincing and plausible.

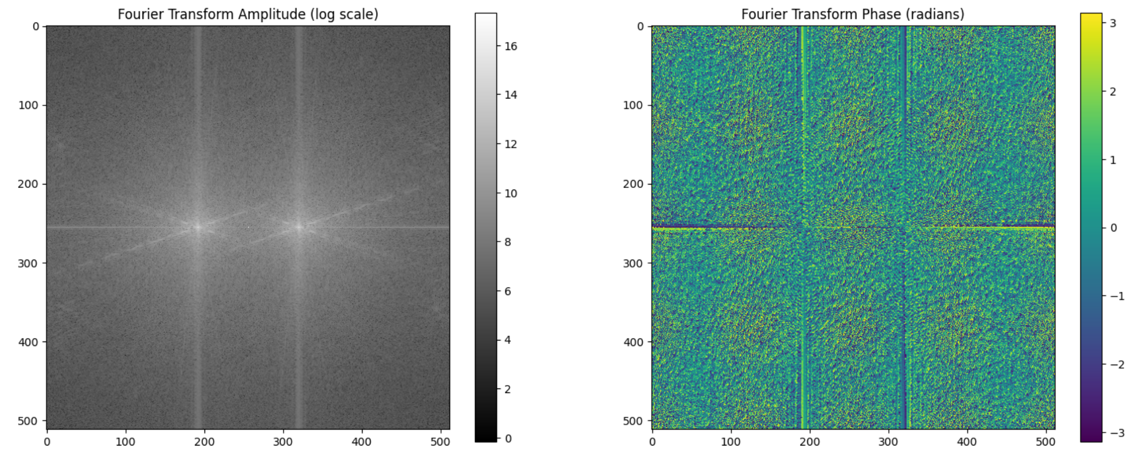
[3]:
```
buster = io.imread('/home/apd/Projects/ImageAnalysis/HW4/
 ↪Buster_Keaton_General_Train_512.png')
buster_fft = np.fft.fft2(buster)
showim(buster)

F_shifted = np.fft.fftshift(buster_fft)
amplitude = np.abs(F_shifted)
phase = np.angle(F_shifted)

plt.figure(figsize=(18,7))
plt.subplot(1,2,1)
plt.title("Fourier Transform Amplitude (log scale)")
plt.imshow(np.log(amplitude), cmap='gray')
plt.colorbar()

plt.subplot(1,2,2)
plt.title("Fourier Transform Phase (radians)")
plt.imshow(phase)
plt.colorbar()
plt.show()
```



[4]:
```
buster = io.imread('/home/apd/Projects/ImageAnalysis/HW4/
 ↪Buster_Keaton_General_Train_512_sineMod.png')
buster_fft = np.fft.fft2(buster)
showim(buster)

F_shifted = np.fft.fftshift(buster_fft)
```

```
amplitude = np.abs(F_shifted)
phase = np.angle(F_shifted)

plt.figure(figsize=(18,7))
plt.subplot(1,2,1)
plt.title("Fourier Transform Amplitude (log scale)")
plt.imshow(np.log(amplitude), cmap='gray')
plt.colorbar()

plt.subplot(1,2,2)
plt.title("Fourier Transform Phase (radians)")
plt.imshow(phase)
plt.colorbar()
plt.show()
```





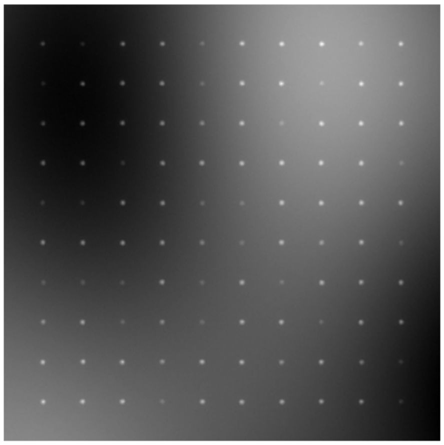
The amplitude array of the original image reveals a central frequency peak. After applying the sine modification, the frequency distribution changes which introduces two distinct peaks (that likely correspond to the positive and negative values of the sine wave). The modification results in a reduction of intensity at the central peak, as most of the original content is affected by the sine modulation. However, traces of the central peak are still slightly visible, even though much of the image information is redistributed.

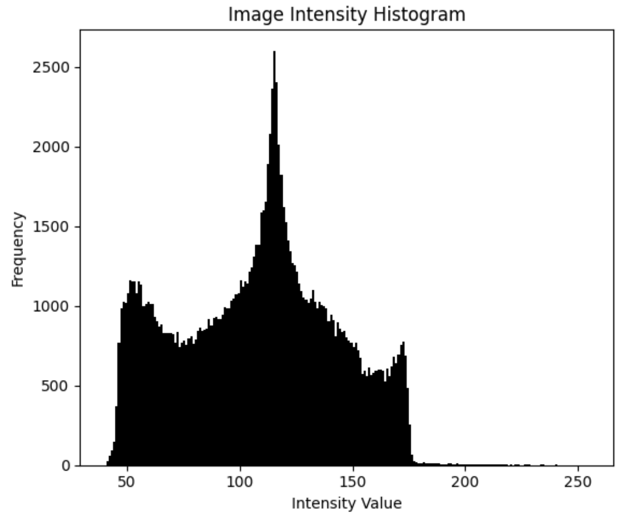
2) Quantized aggregates.

Download "emitters_33px_100ph.png," a simulated image of 100 point-sources arranged on a grid. Each emitter is located at a multiple of (33, 33) pixels from the top left. Imagine that these are images of single molecules, a protein linked to green fluorescent protein for example. You want to know: do these proteins exist as monomers (one unit), dimers (two units), trimers (three units), ..., or a combination of these? In other words, is the brightness of the dots quantized, and if so, how many quanta are there per dot? Is this the same for all of them? Your task is to figure this out from the image

(a) Clearly explain each step of your assessment in addition to stating the answer. (Pasting code without explanations will get zero points.) Except for the placement on a grid, this is a very "real-world" problem – in fact, it was inspired by a graduate student talk I attended!

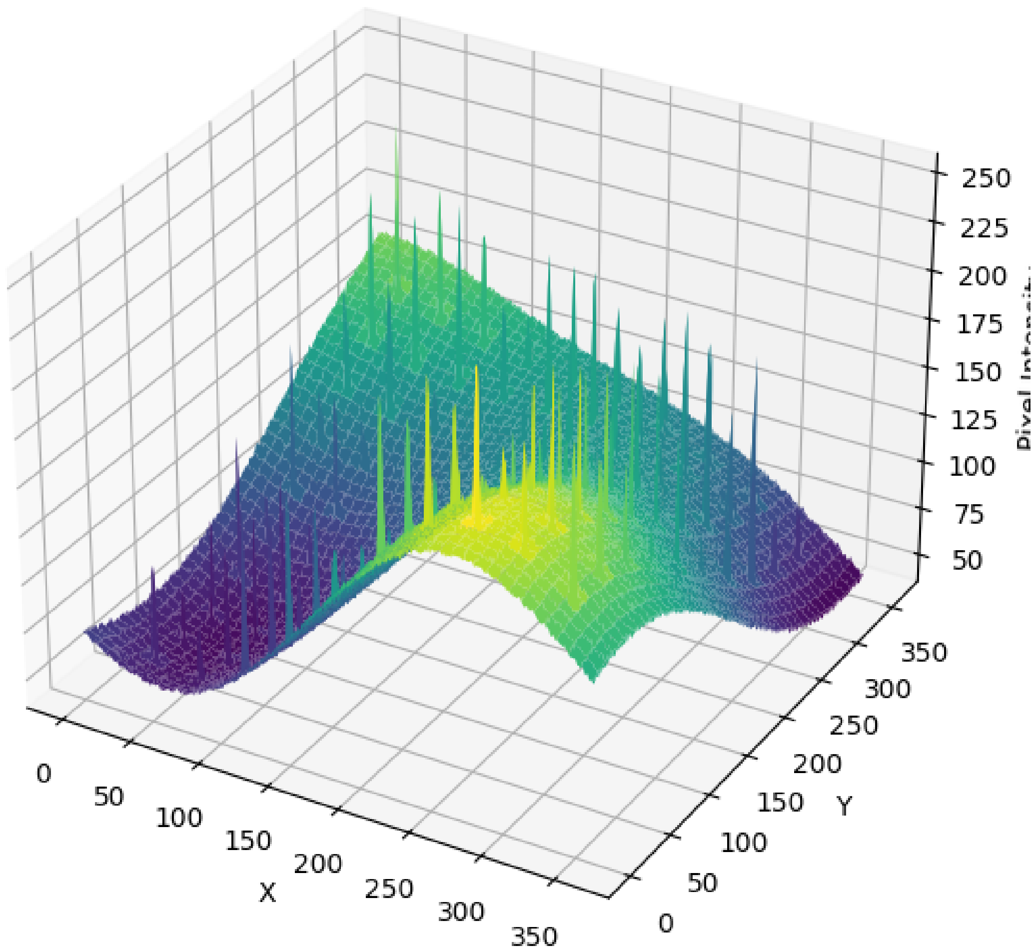
[5]:
```
emitters = io.imread('/home/apd/Projects/ImageAnalysis/HW4/emitters_33px_100ph.
 ↪png')
showim(emitters, show_hist = True)
```



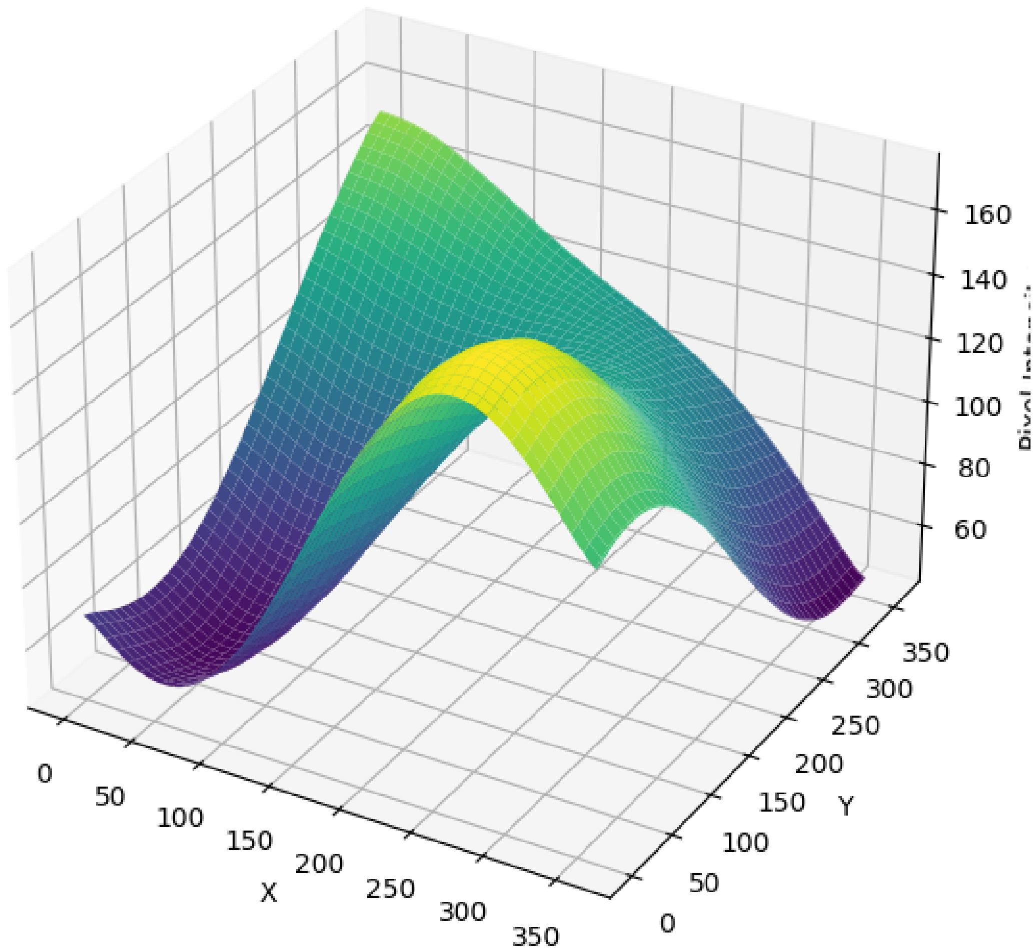


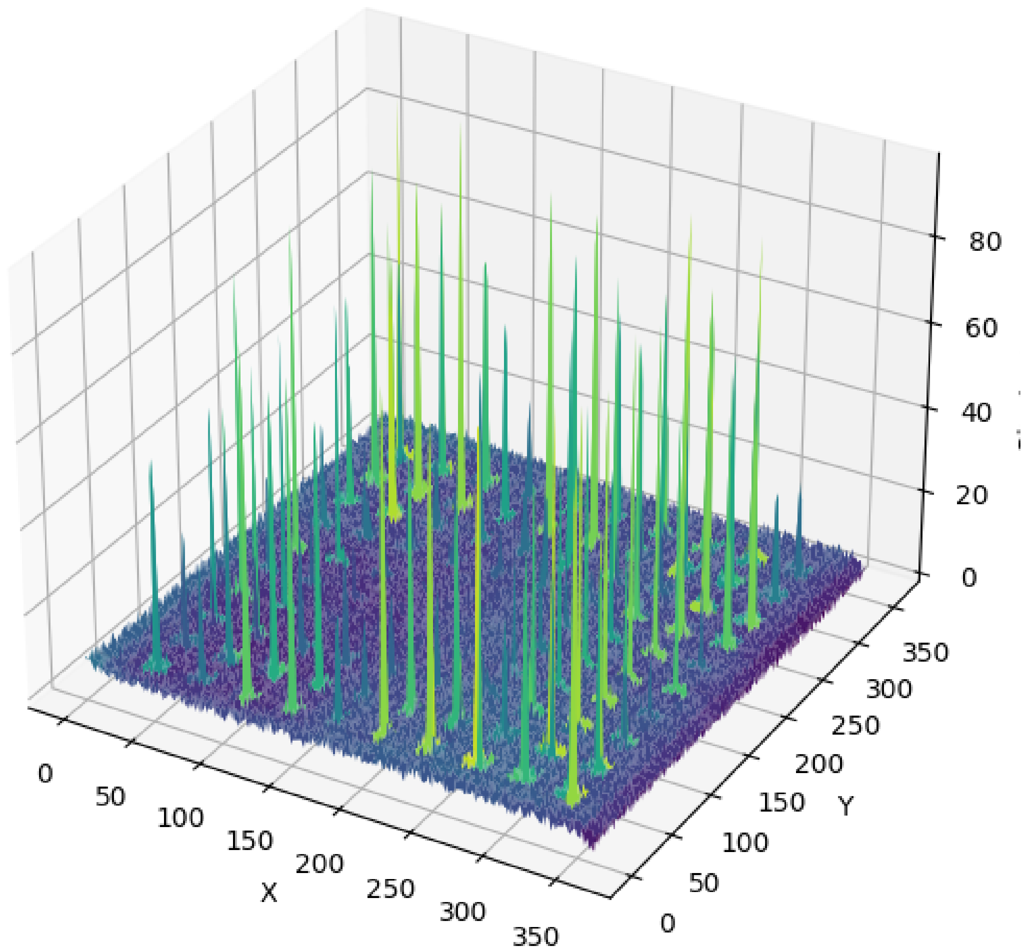
[153]:
```python
def extract_point_sources_max(image, grid_size=10, step=33, margin=33,␣
↪neighborhood_size=3):

    height, width = image.shape

    half_size = neighborhood_size // 2

    indices = np.arange(grid_size)
    x_coords = margin + indices * step
    y_coords = margin + indices * step

    X, Y = np.meshgrid(x_coords, y_coords)

    x_flat = X.flatten().astype(int)
    y_flat = Y.flatten().astype(int)

    valid_mask = (
        (x_flat - half_size >= 0) &
        (x_flat + half_size < width) &
        (y_flat - half_size >= 0) &
        (y_flat + half_size < height)
    )

    x_flat = x_flat[valid_mask]
    y_flat = y_flat[valid_mask]

    max_values = np.empty_like(x_flat, dtype=image.dtype)

    for i, (y, x) in enumerate(zip(y_flat, x_flat)):
```

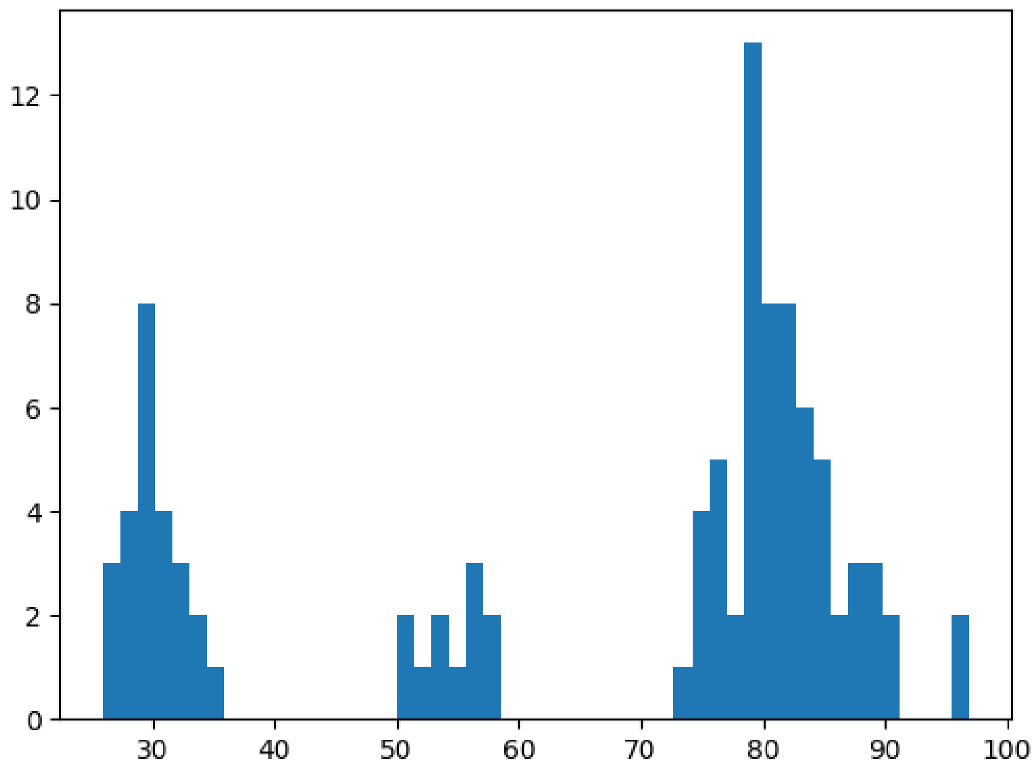
```
        window = image[y - half_size : y + half_size + 1, x - half_size : x +␣
↪half_size + 1]
        max_values[i] = window.max()

    coordinates = np.vstack((y_flat, x_flat)).T

    return max_values, coordinates


A = emitters
surfplot(A)
B = (gaussian(A, sigma=11)*255)
surfplot(B)
C = A - B
C = C - C.min()
surfplot(C)

signals, mask = extract_point_sources_max(C)

bin_width = 1
min_val = np.min(signals)
max_val = np.max(signals)
bins = np.arange(min_val, max_val + bin_width, bin_width)
plt.hist(signals, bins=50);
```



250
225
200
175
150
125
100
75
50
0
50
100
150
200
250
300
350
X
0
50
100
150
200
250
300
350
Y


0
50
100
150
200
250
300
350
X
0
50
100
150
200
250
300
350
Y
60
80
100
120
140
160


0
50
100
150
200
250
300
350
X
0
50
100
150
200
250
300
350
Y
0
20
40
60
80



[154]:
```
monomers = 0
dimers = 0
trimers = 0

for signal in signals:
    if signal > 0 and signal < 40:
        monomers += 1
    elif signal > 45 and signal < 65:
        dimers += 1
    elif signal > 65:
        trimers += 1
print(f"number of monomers: {monomers}")
print(f"number of dimers: {dimers}")
print(f"number of trimers: {trimers}")
```

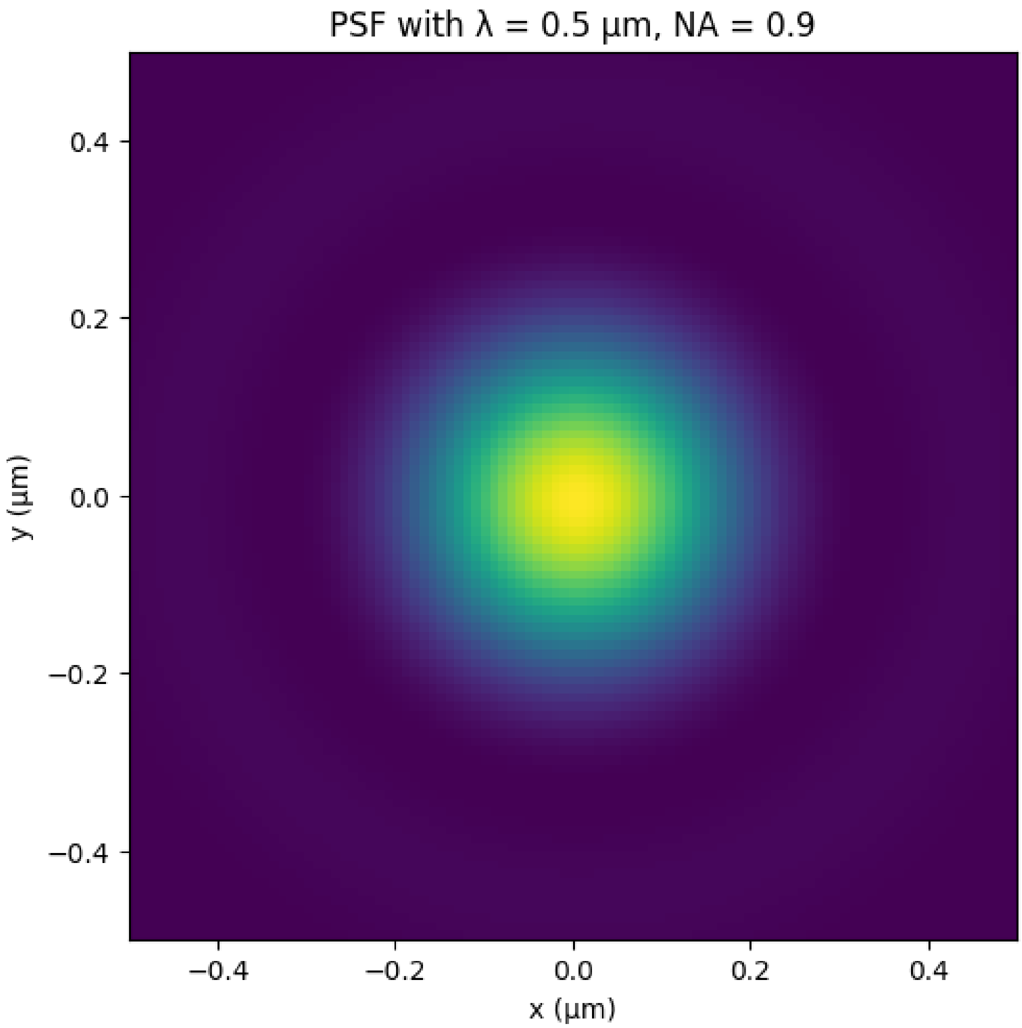
```
number of monomers: 25
number of dimers: 11
number of trimers: 64
```

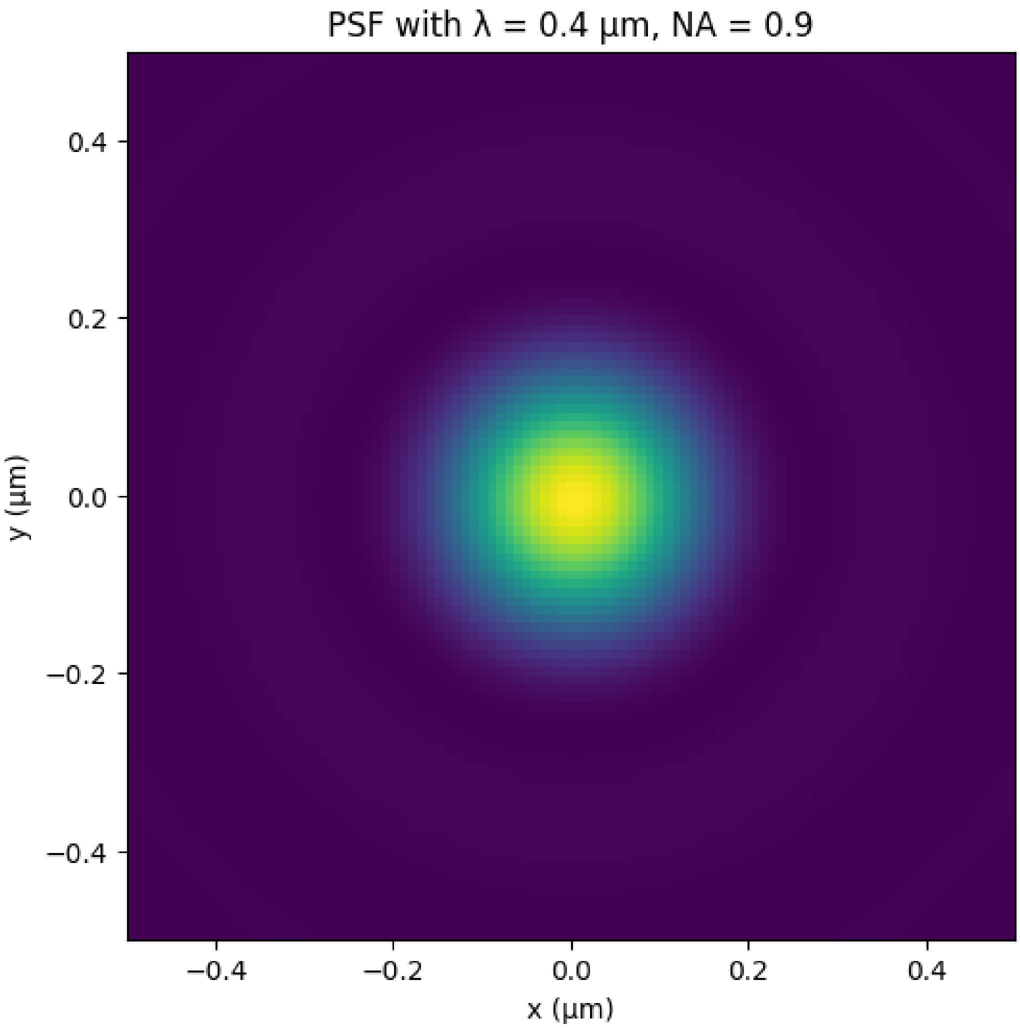
Since the signals are spikes on a wave-like, slow-changing background, I decided to convolve the image with a gaussian convolution to get rid of the spikes, leaving essentially only the background. Then, I picked out the max value of the neighborhood of each signal in the emitter array (each 33 pixels apart) to get the representative signal. After this, I histogrammed the values, displaying three distinct intensity levels from ~25 5o ~35, ~50 to ~60, and ~70 to ~90 (with one extra around 97). From these distinct conncentrations I pulled out the amount of each, with a total of 25 monomers, 11 dimers and 64 trimers. The brightness of the dots seems quantized within these three clusters, indicating the proteins exist either as amonomer, dimer, or trimer.

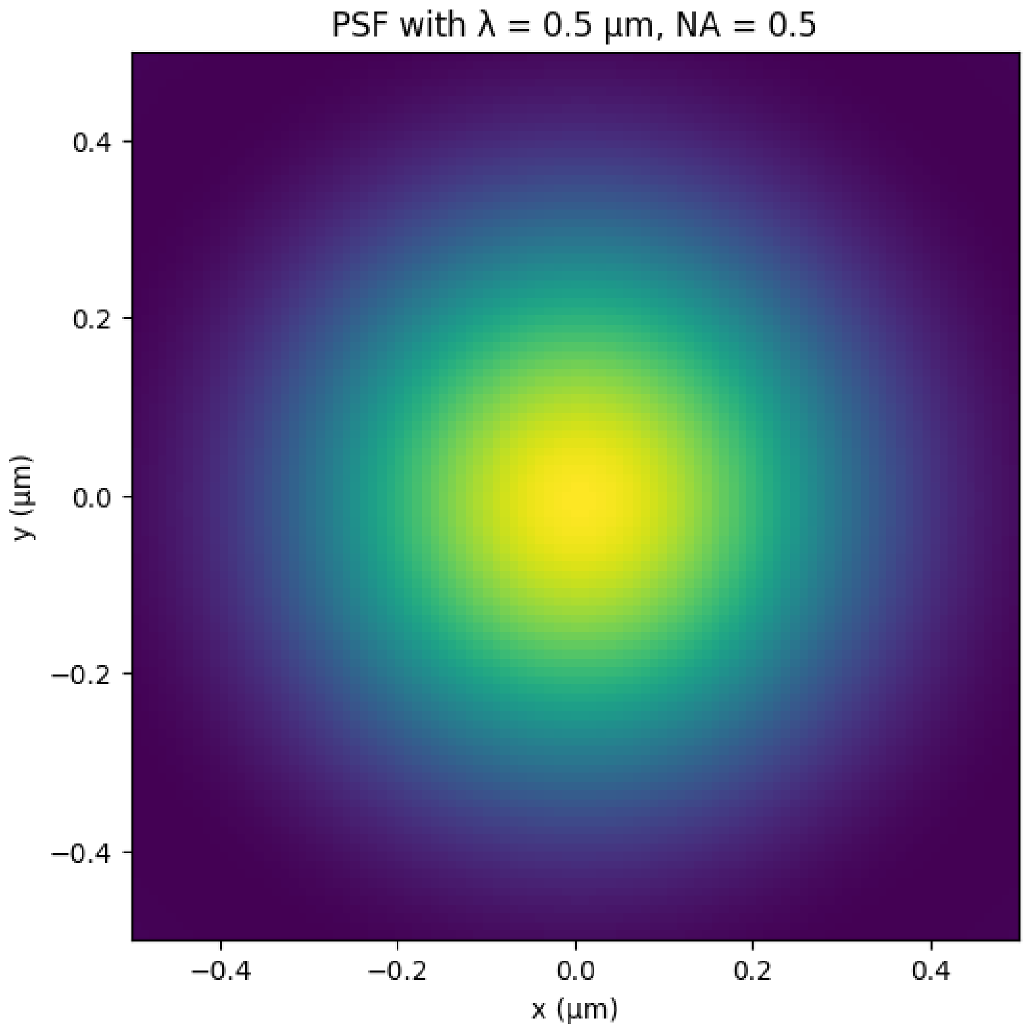
3) Make a PSF

Write a function that calculates the point spread function, outputting it as an N x N array as a function of input parameters: N, the wavelength of light, the numerical aperture, and the pixel scale (i.e. the distance in the focal plane that each pixel corresponds to.)

[155]:
```python
from scipy.special import j1

def calculate_psf(N, wavelength, NA, pixel_scale):

    x = np.linspace(-N//2, N//2, N) * pixel_scale
    y = np.linspace(-N//2, N//2, N) * pixel_scale
    X, Y = np.meshgrid(x, y)
    r = np.sqrt(X**2 + Y**2)

    v = (2 * np.pi / wavelength) * NA * r

    PSF = np.zeros_like(v)
    PSF[v == 0] = 1
    PSF[v != 0] = 4 * (j1(v[v != 0]) / v[v != 0])**2

    return PSF

N = 101
wavelengths = [0.5, 0.4, 0.5]
NA_values = [0.9, 0.9, 0.5]
pixel_scale = 1 / N

for i, (wavelength, NA) in enumerate(zip(wavelengths, NA_values)):
    psf = calculate_psf(N, wavelength, NA, pixel_scale)
    plt.figure(figsize=(6, 6))
    plt.imshow(psf, cmap='viridis', extent=[-0.5, 0.5, -0.5, 0.5])
    plt.title(f"PSF with   = {wavelength}  m, NA = {NA}")
    plt.xlabel('x ( m)')
    plt.ylabel('y ( m)')
    plt.show()
```


PSF with λ = 0.5 μm, NA = 0.9
0.4
0.2
0.0
−0.2
−0.4
y (μm)
−0.4
−0.2
0.0
0.2
0.4
x (μm)

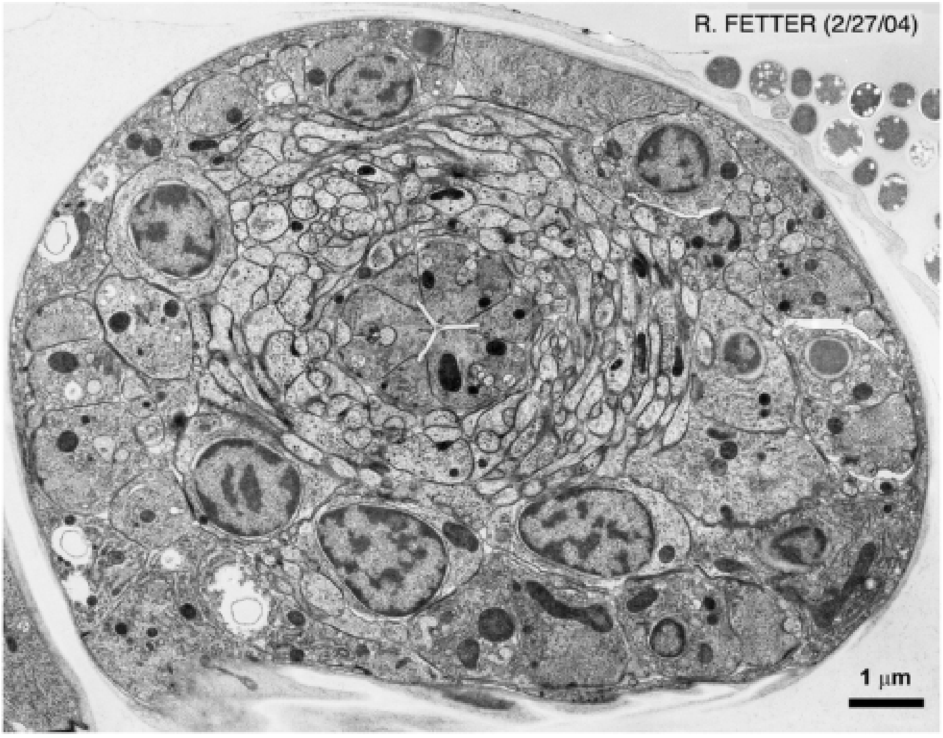

PSF with λ = 0.4 μm, NA = 0.9
y (μm)
0.4
0.2
0.0
−0.2
−0.4
x (μm)
−0.4
−0.2
0.0
0.2
0.4



4. A worse worm image!

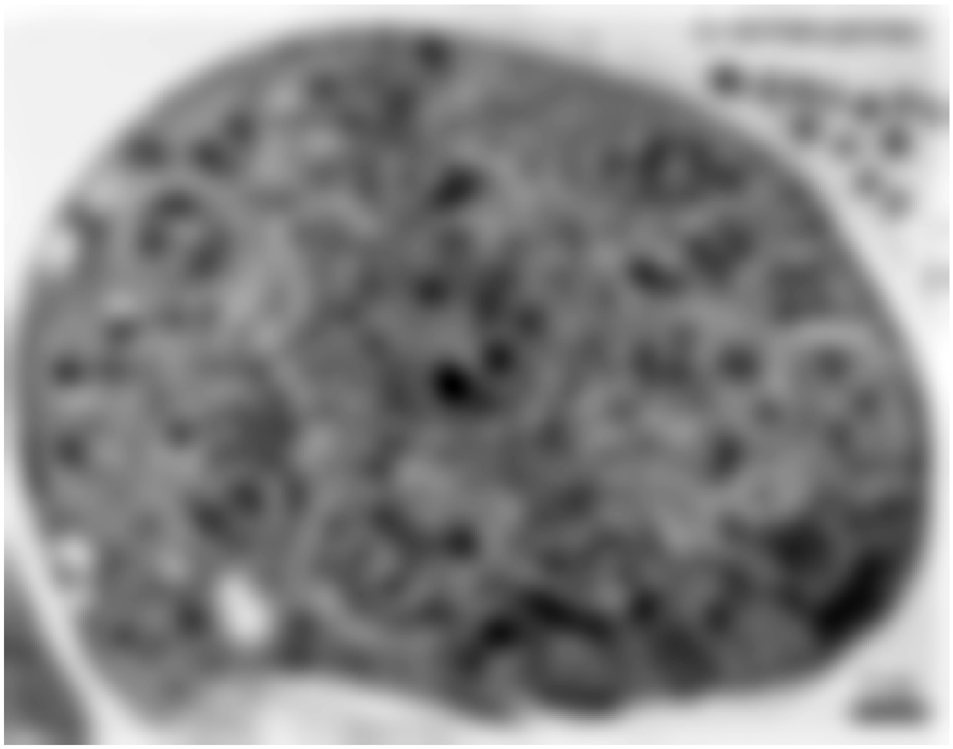
[147]:
```
worm = io.imread('/home/apd/Projects/ImageAnalysis/HW4/
 ↪fetter_Celegans_cellfig10.jpg')
showim(worm)
worm.max()
```

[147]: np.uint8(255)



[150]:
```
lam = 0.53
NA = 0.7
N = 101
pixel_scale = (1/111)

psf = calculate_psf(N, lam, NA, pixel_scale)
psf = psf /psf.sum()

worse_worm = convolve(worm, psf)

showim(worse_worm)
worse_worm.shape
```

[150]: (1080, 1378)



I opened the image in ImageJ and measured the scalebar in pixels, getting around 111 pixels. Thus I set the pixel_scale to 1/111, and set the other parameters as mentioned in the description of the problem. It seemed to just blur the original image, which we expect since TEM's can acheive much higher resolution.

5 SNR and Poisson Noise

For a Poisson-distributed random variable like the total number of photons (Nphoton), how are Nphoton and the signal-to-noise ratio (SNR) related? Remember: "noise" is the standard deviation of whatever we're measuring. Yes, this is a very short question.

The signal to noise ratio is related to the number of photons (if it is a posson-distributed random variable) by square root, or in other words SNR ~ sqrt(N__photons), since the noise is just the square root of the mean, which in this case is just N__photons

6: My gameplan

To tackle this problem, the first thing I need to do is adjust the scale of the PSF function I used earlier. Before, I was working with a fine scale of 0.01 m/px, but now I have to account for a larger scale of 0.1 m/px to match a typical camera resolution. The idea here is to essentially create a "nested" grid, where each camera pixel represents a larger block of fine-grid pixels. For instance, with the 0.1 m/px camera scale, each camera pixel will correspond to 10x10 fine-grid pixels. I'm thinking I'll need to sum or average the intensity values from these smaller grid points into each camera pixel, kind of like a downscaling process. Once that's done, it will effectively pixelate the image to match the lower resolution a camera would capture.

After that, I'll need to scale the PSF output to match a specific photon count, N_photon. The PSF gives me the probability of photons arriving at different points, but since photon arrival is random, the actual number of photons at each pixel should follow a Poisson distribution. So, I'll modify the function to generate Poisson-distributed photon counts at each pixel, using the scaled PSF values as the mean. This part should be pretty straightforward and will just require adding one line of code to introduce the Poisson noise. Once that's in place, I'll be able to run the function with two examples for N_photon = 50 and two more for N_photon = 500. The final function will take in the necessary inputs like N_photon, lambda, NA, the camera scale, and the fine grid scale, and it will output the pixelated image with noise.