Homework 7 Aiden Dillon

0. Importing and Defining

In [145…

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.special import j1
from tqdm.auto import tqdm
from skimage import io
from scipy.optimize import minimize
from scipy.ndimage import convolve
from skimage import restoration


def showim(im_array, figsize=(4, 4), show_hist=False, nbins=None, bin_width=None, c

    if isinstance(im_array, (list, tuple)):
        n_images = len(im_array)
        fig_width, fig_height = figsize
        plt.figure(figsize=(fig_width * n_images, fig_height))

        for idx, img in enumerate(im_array):
            plt.subplot(1, n_images, idx + 1)
            plt.imshow(img, cmap=cmap, vmin=vmin, vmax=vmax)

            if titles and isinstance(titles, (list, tuple)) and len(titles) == n_im
                plt.title(titles[idx])
            elif titles and isinstance(titles, str):
                plt.title(titles)

            plt.axis('off')
        plt.tight_layout()

        plt.show()
    else:
        plt.figure(figsize=figsize)

        if show_hist:
            plt.subplot(1, 2, 1)
            plt.imshow(im_array, cmap=cmap, vmin=vmin, vmax=vmax)

            if titles and isinstance(titles, str):
                plt.title(titles)

            plt.axis('off')
            plt.subplot(1, 2, 2)

            im_flattened = im_array.ravel()
            min_val = np.floor(im_flattened.min())
            max_val = np.ceil(im_flattened.max())
```

```
            if bin_width is not None:
                bins = np.arange(min_val, max_val + bin_width, bin_width)
            elif nbins is not None:
                bins = nbins
            else:
                bins = int(max_val - min_val)

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

        else:
            plt.imshow(im_array, cmap=cmap, vmin=vmin, vmax=vmax)

            if titles and isinstance(titles, str):
                plt.title(titles)

            plt.axis('off')
        plt.tight_layout()
        plt.show()


def sim_ps(N_camera, wavelength, NA, camera_scale, fine_scale, N_photon, center = N
    block_size = int(camera_scale / fine_scale)
    fine_grid_N = N_camera * block_size

    x = np.linspace(-fine_grid_N//2, fine_grid_N//2, fine_grid_N) * fine_scale
    y = np.linspace(-fine_grid_N//2, fine_grid_N//2, fine_grid_N) * fine_scale
    X, Y = np.meshgrid(x, y)
    if center:
        xc, yc = center
    else: xc, yc = 0, 0

    r = np.sqrt((X - xc)**2 + (Y - yc)**2)

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

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

    psf_fine = psf_fine / psf_fine.sum()

    psf_camera = np.zeros((N_camera, N_camera))

    for i in range(N_camera):
        for j in range(N_camera):
            block = psf_fine[i*block_size:(i+1)*block_size, j*block_size:(j+1)*bloc
            psf_camera[i, j] = np.sum(block)

    psf_camera /= np.sum(psf_camera)

    psf_camera *= N_photon

    noisy_psf = np.random.poisson(psf_camera)
```

```
    background = np.random.poisson(B, (N_camera, N_camera))

    final_image = noisy_psf + background


    return final_image


def calculate_MLE(intensities, initial_guess, camera_scale=0.01, origin_center=True
    N = intensities.shape[0]
    x = np.arange(N)
    y = np.arange(N)

    if origin_center==True:
        x = x - (N - 1) / 2
        y = y - (N - 1) / 2

    X, Y = np.meshgrid(x, y)

    def objective(params, X, Y, data):
        xc, yc, A0, sigma, B = params
        model = A0 * np.exp(-((X - xc)**2 + (Y - yc)**2) / (2 * sigma**2)) + B
        return np.sum(model - data * np.log(np.where(model > 0, model, 1)))

    results = minimize(objective, initial_guess, args=(X, Y, intensities))
    xc_est, yc_est, A0_est, sigma_est, B_est = results.x

    if camera_scale:
        xc_est *= camera_scale
        yc_est *= camera_scale

    return xc_est, yc_est, A0_est, sigma_est, B_est


def calculate_center_RMSE(calculated_positions, true_positions):

    calculated_positions = np.array(calculated_positions)
    true_positions = np.array(true_positions)

    if calculated_positions.ndim == 1:
        calculated_positions = calculated_positions.reshape(1, -1)
    if true_positions.ndim == 1:
        true_positions = true_positions.reshape(1, -1)

    squared_diffs = (calculated_positions - true_positions) ** 2

    rms_error = np.sqrt(np.mean(np.sum(squared_diffs, axis=1)))

    return rms_error


def calculate_image_RMSE(original, deconvolved):
    difference = original - deconvolved
    squared_diff = np.square(difference)
    mean_squared_diff = np.mean(squared_diff)
```

```
    rms_error = np.sqrt(mean_squared_diff)
    return rms_error
```

1. Gaussian MLE and number of photons

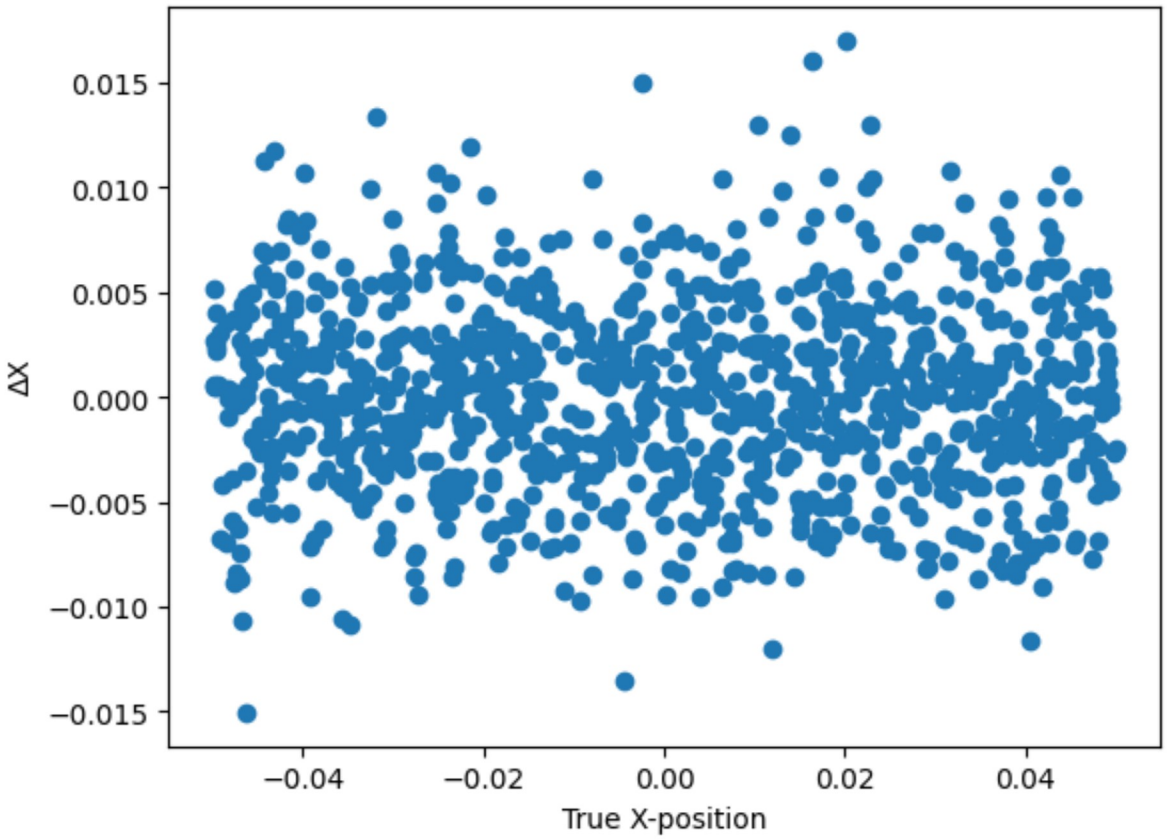
(a)

As in HW5 and HW6, simulate images of a point source: 7 × 7 px images with scale = 100 nm/px (i.e. 0.1 µm/px), λ = 510 nm, NA = 0.9, background mean = 10. The true center (x_0, y_0) should be a random number from a uniform distribution between -0.05 µm and 0.05 µm, that is [-0.5, 0.5] pixels, in each dimension.

For Nphoton = 1000, calculate the error in x_{MLE}, Δx = x_{MLE}-x_0, where (x_{MLE} , y_{MLE}) is your MLE estimate of the particle position. Plot Δx vs. the true x_0. Is there bias? (Just assess this by eye – you don't need to quantify any bias.)

In [ ]:
```
N_camera = 7
wavelength = 0.510
NA = 0.9
N_photons = 1000
camera_scale = 0.1
fine_scale = 0.001
bg = 10

m = 1000
initial_guess = [0,0,100,1,0]

true_x_list = []
x_error_list = []


for _ in range(m):

    center_x = np.random.uniform(-0.05,0.05)
    center_y = np.random.uniform(-0.05,0.05)
    current_center = (center_x,center_y)

    image = image = sim_ps(N_camera=N_camera, wavelength=wavelength, NA=NA, camera_
            fine_scale=fine_scale, N_photon=N_photons, center=current_center, B=bg)

    MLE = calculate_MLE(image, initial_guess=initial_guess, camera_scale=camera_sca

    x_MLE = MLE[0]
    x_error = x_MLE - current_center[0]

    true_x_list.append(center_x)
    x_error_list.append(x_error)

plt.scatter(true_x_list,x_error_list)
plt.xlabel('True X-position (µm)')
plt.ylabel('ΔX (µm)');
```

Out[ ]: Text(0, 0.5, 'ΔX')



It seems as though there is no bias, with the error being at max around 0.015 µm in magnitude.

(b)

Consider a number of photons Nphoton from 40 to 40,000, making a logarithmically spaced set of at least 10 N_{photon} values in this range.

For each Nphoton value, make M = 100 images again with the true center (x_0, y_0) being a random number uniform over [-0.5, 0.5] in each dimension.

Calculate RMS error of the MLE localization (i.e. the square root of the average of ($\Delta x^2 + \Delta y^2$)).

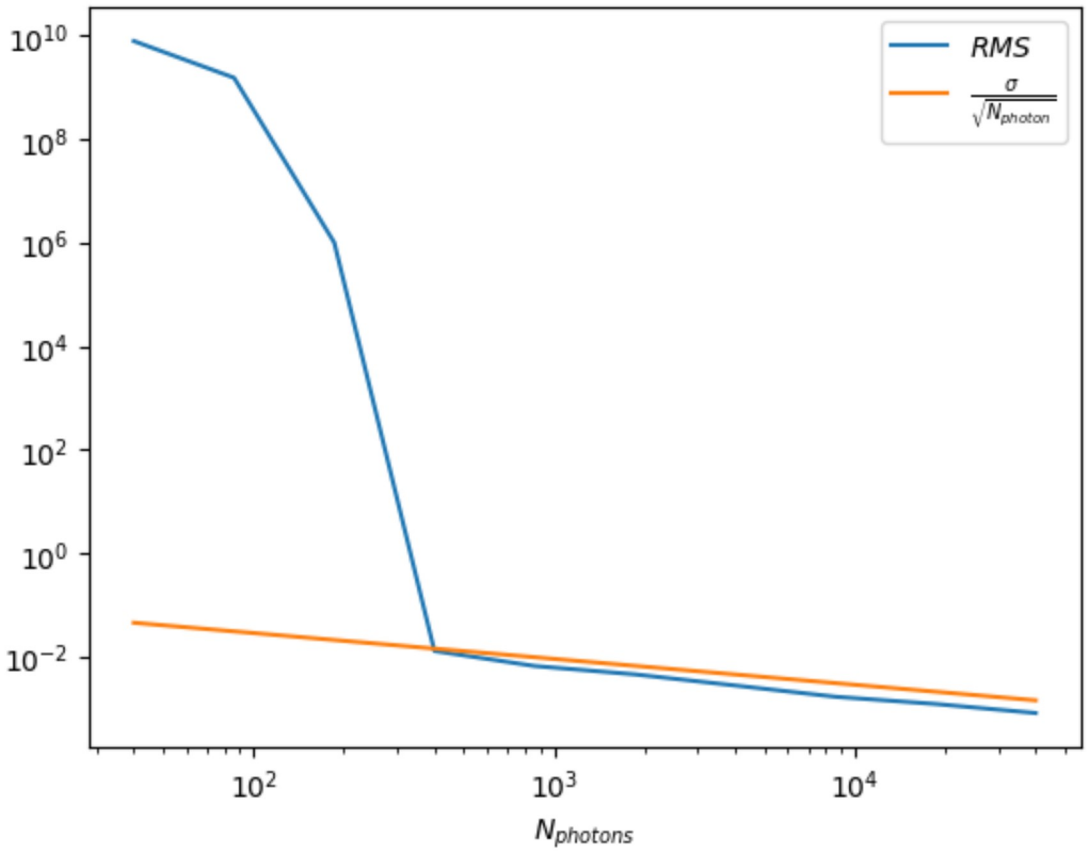
Submit a log-log plot of RMS error vs. Nphoton. Include on the graph a plot of $\frac{\sigma}{\sqrt{N_{photon}}}$ vs. N_{photon}, where $\sigma = \frac{\lambda}{2(NA)}$

In [60]:
```
N_camera = 7
wavelength = 0.510
NA = 0.9
camera_scale = 0.1
fine_scale = 0.001
bg = 10

m = 100
```

```
N_photons_list = np.logspace(np.log10(40), np.log10(40000), num=10)
RMS_list = []
value_list = []

sigma = wavelength/(2*NA)

for N_photons in tqdm(N_photons_list):

    current_N_photons = N_photons

    true_positions_list = []
    estimated_positions_list = []

    for _ in range(m):

        center_x = np.random.uniform(-0.05,0.05)
        center_y = np.random.uniform(-0.05,0.05)
        current_center = (center_x,center_y)

        image = image = sim_ps(N_camera=N_camera, wavelength=wavelength, NA=NA, cam
                fine_scale=fine_scale, N_photon=current_N_photons, center=current_c

        MLE = calculate_MLE(image, initial_guess=initial_guess, camera_scale=camera

        true_positions_list.append(current_center)
        estimated_positions_list.append((MLE[0], MLE[1]))

    RMS = calculate_center_RMSE(estimated_positions_list, true_positions_list)
    RMS_list.append(RMS)

    value = sigma / np.sqrt(current_N_photons)
    value_list.append(value)

plt.loglog(N_photons_list, RMS_list, label = r'$RMS$')
plt.loglog(N_photons_list, value_list, label = r'$\frac{σ}{\sqrt{N_{photon}}}$')
plt.xlabel(r'$N_{photons}$')
plt.legend()
```

```
  0%|          | 0/10 [00:00<?, ?it/s]
```

Out[60]: <matplotlib.legend.Legend at 0x7fcaca7b6710>



(c)

Explain why comparing the RMS error to $\frac{\sigma}{\sqrt{N_{photon}}}$ *makes sense.*

Comparing the RMS error to $\frac{\sigma}{\sqrt{N_{\text{photon}}}}$ lets us to assess the effectiveness of the MLE method and how closely it approaches the theoretical (diffraction) limit. This comparison indicates whether the localization precision achieved is as close as possible to what is theoretically expected, assuming photon shot noise is the primary source of error.

2. Radial-symmetry-based particle localization. (5 pts.)

Test the radial symmetry based localization method I noted in class. You don't have to write it yourself – I'll supply a function; see below. As in #1, test this with simulated images.

(a)

Time how long per image the radial-symmetry localization takes. (Again, don't count the image creation time.) State the result, and compare to your results for centroids and MLE from the previous assignment

In [25]:
```
from radialcenter_ImAnClass import radialcenter
import time

N_camera = 7
```

```
wavelength = 0.51
NA = 0.9
camera_scale = 0.1
fine_scale = 0.01
N_photons = 1000
bg = 10

M = 100

ps_stack = []

for _ in range(M):

    center_x = np.random.uniform(-0.05,0.05)
    center_y = np.random.uniform(-0.05,0.05)
    center = (center_x,center_y)
    new_ps = sim_ps(N_camera=N_camera, wavelength=wavelength, NA=NA, camera_scale=c
              fine_scale=fine_scale, N_photon=N_photons, center = center, B=bg)
    ps_stack.append(new_ps)

ps_stack = np.array(ps_stack)

start_time = time.time()

for i in range(M):

    xRS, yRS = radialcenter(ps_stack[i])

end_time = time.time()

print(f"Average time per radial center calculation: {(end_time-start_time)/M:.6f} s
```

```
Average time per radial center calculation: 0.000389 seconds
```

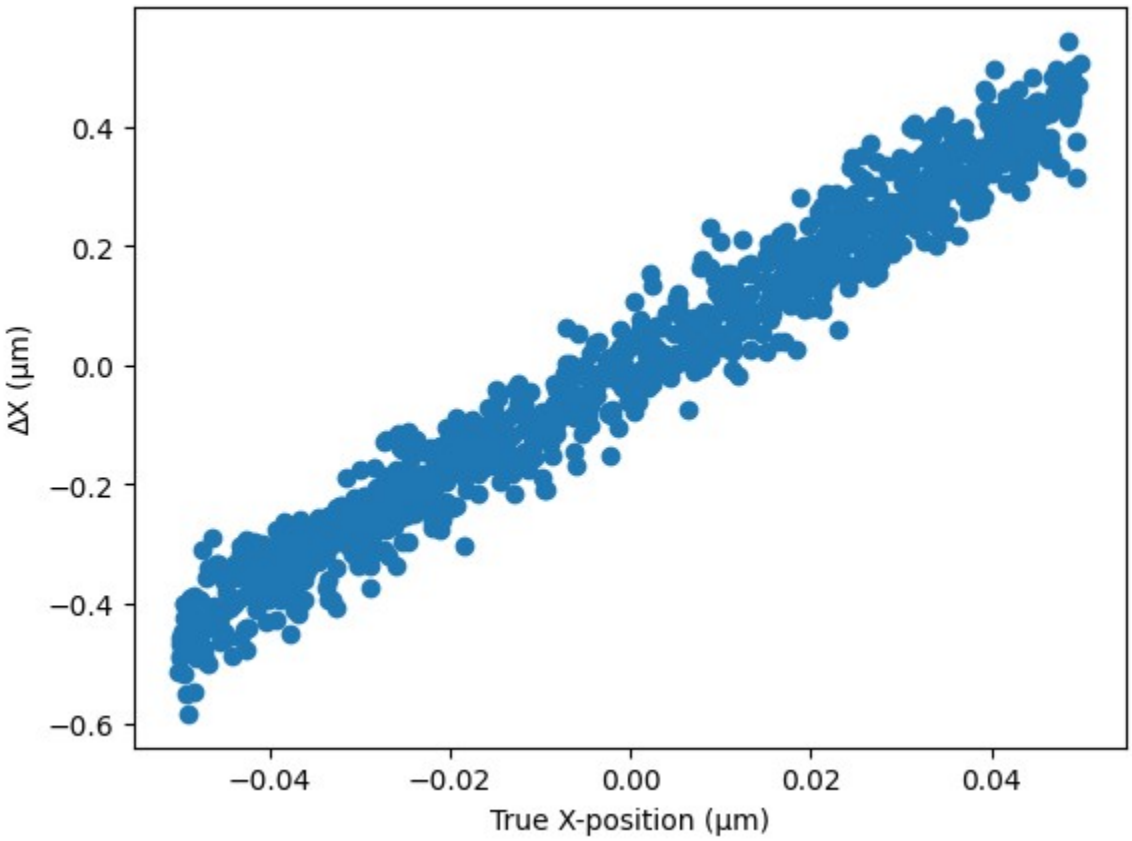
This is faster than the MLE method but slower than centroid (about an order of magnitude slower and faster, respectively).

(b)

Repeat #1a, ...calculate the error in x_c, $\Delta x = x_c\text{-}x_0$. Plot Δx vs. the true x_0. Is there bias?

In [28]:
```
N_camera = 7
wavelength = 0.510
NA = 0.9
N_photons = 1000
camera_scale = 0.1
fine_scale = 0.001
bg = 10

m = 1000
initial_guess = [0,0,100,1,0]

true_x_list = []
x_error_list = []
```

```
for _ in range(m):

    center_x = np.random.uniform(-0.05,0.05)
    center_y = np.random.uniform(-0.05,0.05)
    current_center = (center_x,center_y)

    image = image = sim_ps(N_camera=N_camera, wavelength=wavelength, NA=NA, camera_
            fine_scale=fine_scale, N_photon=N_photons, center=current_center, B=bg)

    xRS, yRS = radialcenter(image)

    x_error = xRS - current_center[0]

    true_x_list.append(center_x)
    x_error_list.append(x_error)

plt.scatter(true_x_list,x_error_list)
plt.xlabel('True X-position (μm)')
plt.ylabel('ΔX (μm)');
```



There dooes seem to be bias with the radial center method, as the error has a clear, linear relationship with the true x-position.

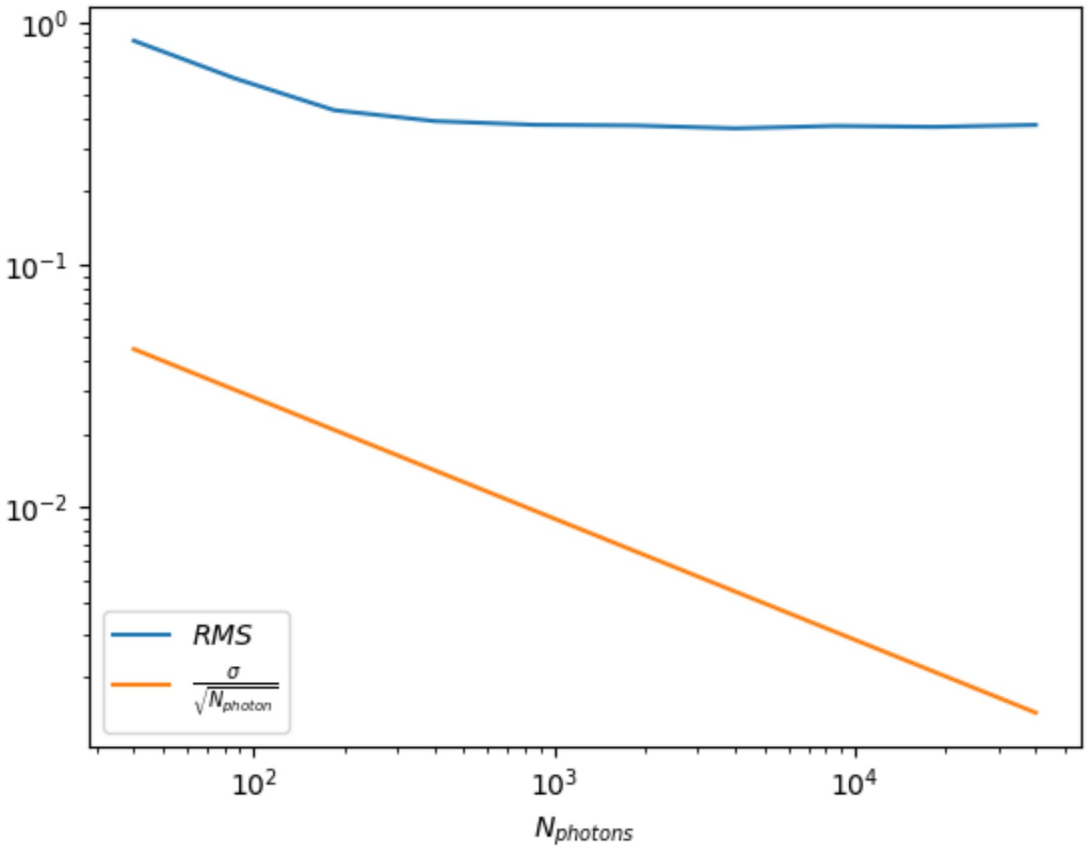
(c)

Repeat#1b, ... Nphoton from 40 to 40,000 ... Calculate RMS error o... Submit a log-log plot of RMS error vs. Nphoton. Include on the graph a plot of $\frac{\sigma}{\sqrt{N_{photon}}}$ *vs. Nphoton, where* $\sigma = \frac{\lambda}{2(NA)}$

In [ ]:
```
N_camera = 7
wavelength = 0.510
NA = 0.9
camera_scale = 0.1
fine_scale = 0.001
bg = 10

m = 100

N_photons_list = np.logspace(np.log10(40), np.log10(40000), num=10)
RMS_list = []
value_list = []

sigma = wavelength/(2*NA)

for N_photons in tqdm(N_photons_list):

    current_N_photons = N_photons

    true_positions_list = []
    estimated_positions_list = []

    for _ in range(m):

        center_x = np.random.uniform(-0.05,0.05)
        center_y = np.random.uniform(-0.05,0.05)
        current_center = (center_x,center_y)

        image = image = sim_ps(N_camera=N_camera, wavelength=wavelength, NA=NA, cam
                fine_scale=fine_scale, N_photon=current_N_photons, center=current_c

        xRS, yRS = radialcenter(image)

        true_positions_list.append(current_center)
        estimated_positions_list.append((xRS, yRS))

    RMS = calculate_center_RMSE(estimated_positions_list, true_positions_list)
    RMS_list.append(RMS)

    value = sigma / np.sqrt(current_N_photons)
    value_list.append(value)

plt.loglog(N_photons_list, RMS_list, label = r'$RMS$')
plt.loglog(N_photons_list, value_list, label = r'$\frac{σ}{\sqrt{N_{photon}}}$')
plt.xlabel(r'$N_{photons}$')
plt.legend();
```

```
0%|          | 0/10 [00:00<?, ?it/s]
```

Out[ ]: <matplotlib.legend.Legend at 0x7fcad0ea1c30>



3) Assessing deconvolution (15 pts.)

If we take an image, convolve it with a PSF, and deconvolve it, how well do we recover the original image? Let's try this, using the electron microscope image "mouse_glial_cells_RBurdan_crop.png" that I've posted on Canvas1. This is just a 2D image, not 3D, but the idea would be the same for assessing 3D deconvolution. (It would be even slower, however, to calculate in 3D.) We'll pretend the original image is "true," convolve with a Gaussian PSF, incorporate noise, and then deconvolve with either the same Gaussian PSF or one that's "wrong."

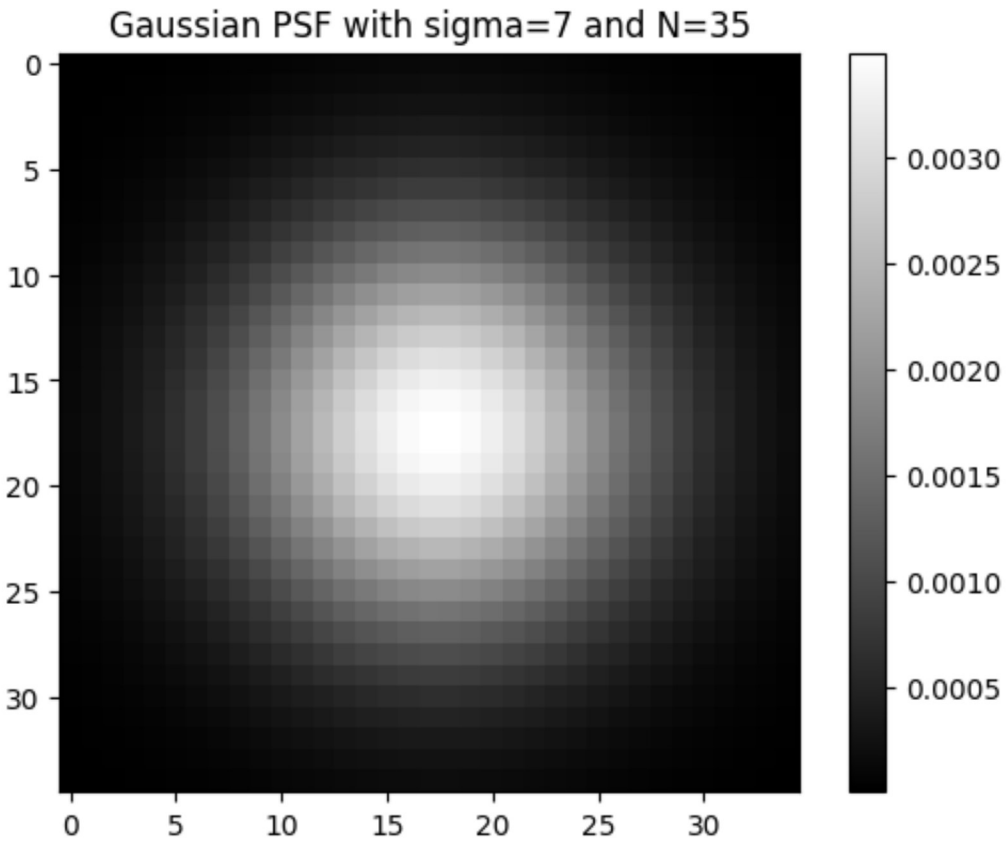
(a)

Write a function that returns an NxN PSF array whose intensity is a Gaussian function of distance to the center pixel, with width sigma. Yes, this should look familiar!

Submit an image of the PSF with sigma=7 and N = 35

In [34]:
```
def gaussian_psf(N, sigma):
    x = np.linspace(-N // 2, N // 2, N)
    y = np.linspace(-N // 2, N // 2, N)
    X, Y = np.meshgrid(x, y)
    distance = np.sqrt(X**2 + Y**2)
    psf = np.exp(-(distance**2) / (2 * sigma**2))
    psf /= psf.sum()
```

```
    return psf

N = 35
sigma = 7

psf = gaussian_psf(N, sigma)
plt.imshow(psf, cmap='gray')
plt.title("Gaussian PSF with sigma=7 and N=35")
plt.colorbar()
plt.show()
```

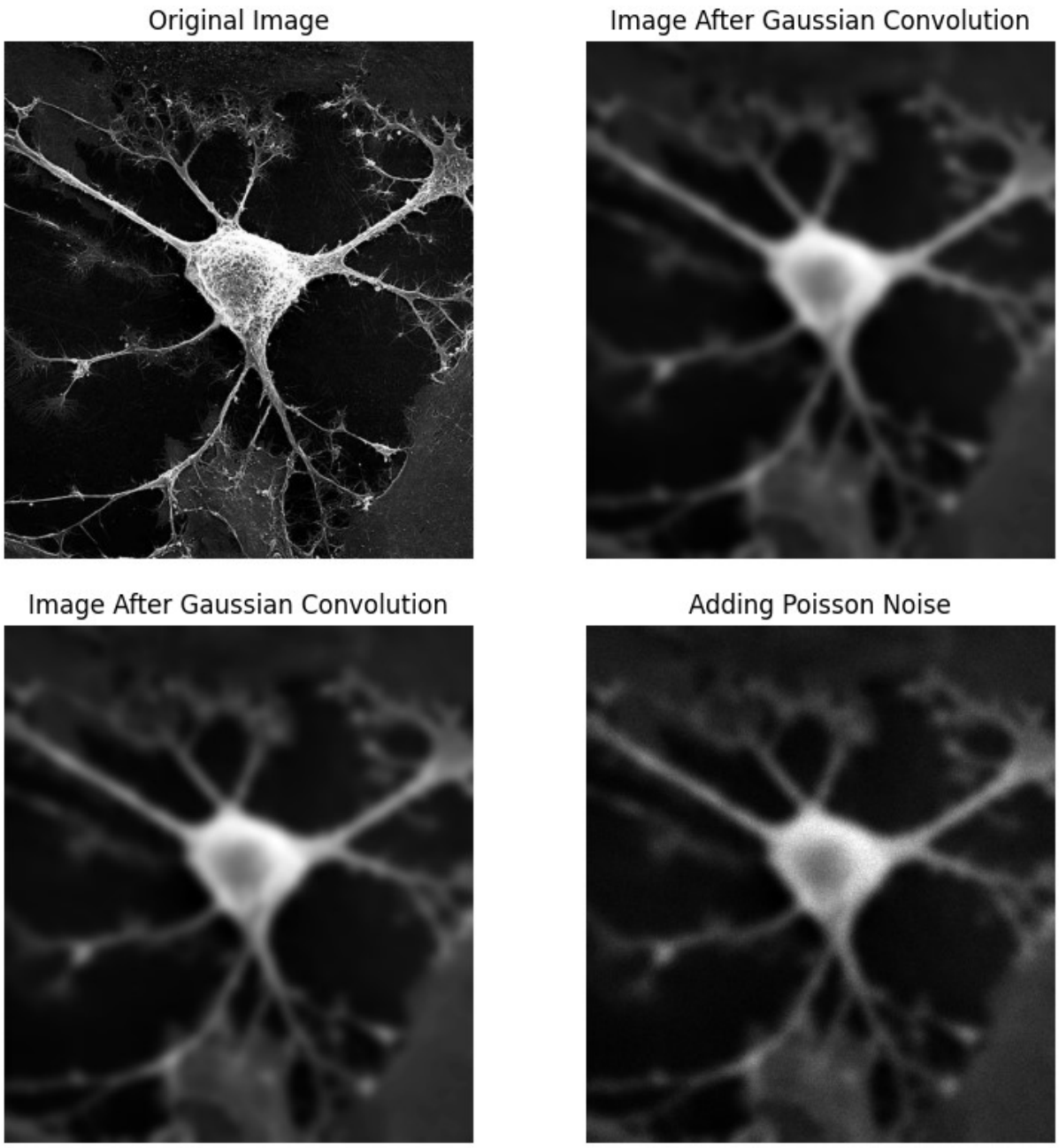


(b)

Load the original image. If its intensity range is 0-1, multiply by 255 to make the range 0-255.

Convolve the image with your Gaussian PSF with sigma=7. (See HW3.) Then replace each pixel with a Poisson random variable of the same mean as the original pixel intensity value (0-255) – i.e. we're pretending that the intensity scale "is" the number of photons. Submit the resulting image.

In [ ]:
```
im = io.imread("/home/apd/Projects/ImageAnalysis/HW7/mouse_glial_cells_RBurdan_crop

im_gaussian_psf = convolve(im, psf)
showim((im, im_gaussian_psf), titles = ('Original Image', 'Image After Gaussian Con

im_noisy = np.clip(np.random.poisson(im_gaussian_psf), 0, 255).astype(np.uint8)
showim((im_gaussian_psf,im_noisy), titles = ('Image After Gaussian Convolution', 'A
```



(c)

• Deconvolve! See the note below on "Implementing deconvolution in Python." Use the Richardson-Lucy algorithm with 20 iterations and make sure you get something sensible. (Look at the image.)

• Calculate the RMSE (root mean square error) between the deconvolved image and the original image.

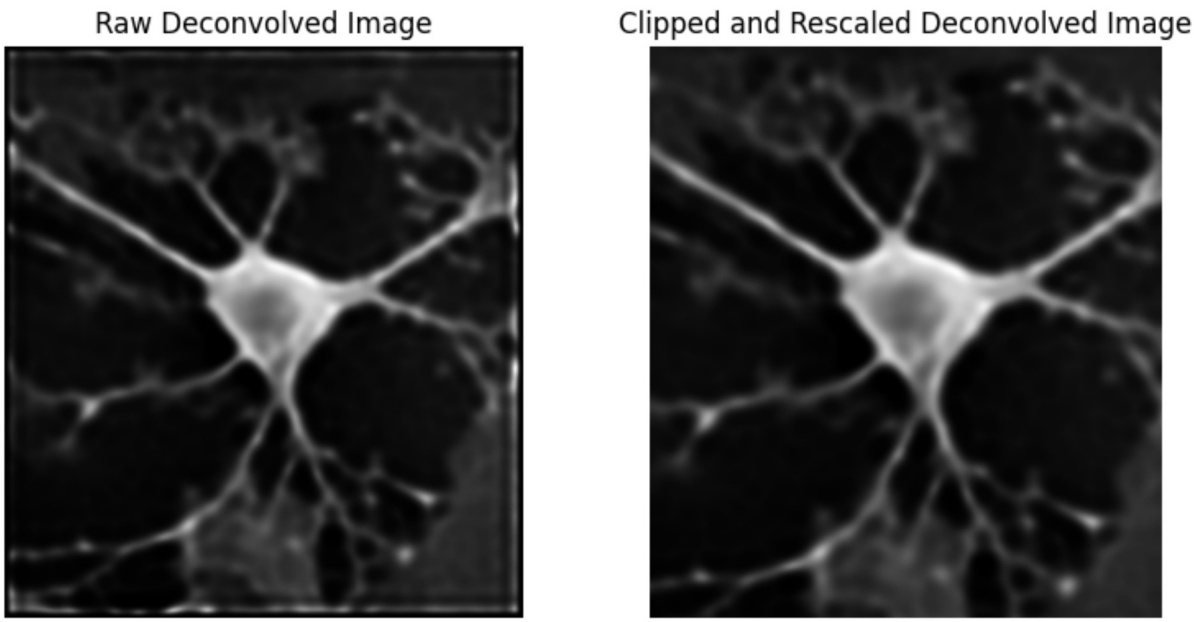
Unfortunately, as you'll see from the image, we get "ringing" near the edges and an overall change of scale, so it's not straightforward to calculate the RMSE. Do the following:

(i) Consider a "cropped" version of the original and the deconvolved images, ignoring the N pixels close to each edge. (Hint: what elements of array x does "x[4:-2]" return?)

(ii) Rescale the intensity of the cropped, deconvolved image so its min and max are equal to the min and max of the cropped original image (which are 0-255).

· Finally, calculate the RMSE between the images, and submit the RMSE value and the deconvolved image.

In [ ]:
```
iterations = 20

im_deconvolved = restoration.richardson_lucy(im_noisy, psf, num_iter=iterations, cl

n_clip = 40 # I found 40 to be able to deal with the iterations = 205 case

im_deconvolved_clipped = im_deconvolved[n_clip:-n_clip, n_clip:-n_clip]
im_deconvolved_clipped_rescaled = 255 * (im_deconvolved_clipped - np.min(im_deconvo
im_deconvolved_clipped_rescaled = im_deconvolved_clipped_rescaled.astype(np.uint8)

RMSE_im_deconvolved = calculate_image_RMSE(im, im_deconvolved)
RMSE_im_deconvolved_clipped = calculate_image_RMSE(im[n_clip:-n_clip, n_clip:-n_cli

showim((im_deconvolved, im_deconvolved_clipped_rescaled), titles = ("Raw Deconvolve

print(f"Raw deconvolved RMSE: {RMSE_im_deconvolved:.3f} || Clipped and rescaled dec
```




```
Raw deconvolved RMSE: 30.241 || Clipped and rescaled deconvolved RMSE: 7.818
```

(d)

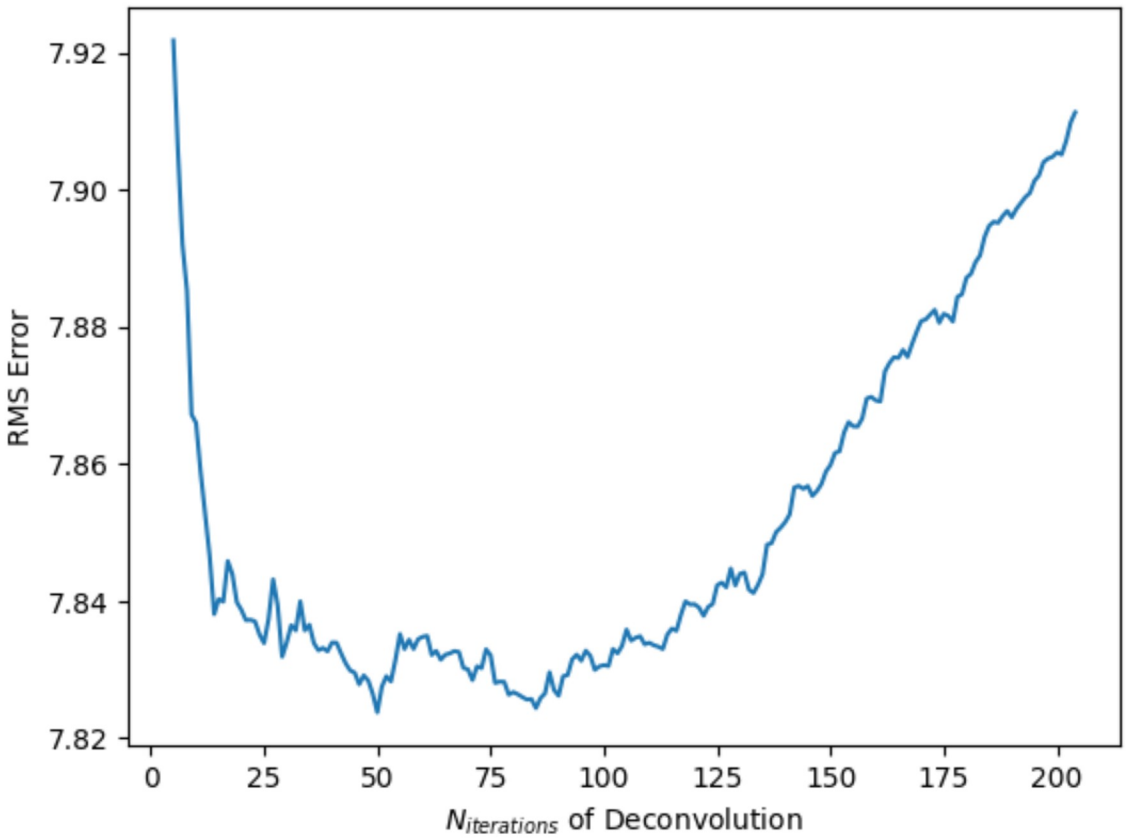
Calculate the RMSE for a number of iterations 5, 15, 25, ..., 205

Submit a plot of RMSE vs. number of iterations, and comment. Note: This will take many minutes to run!

In [118…
```
iterations_list = np.arange(5,205,1)

RMSE_list = []
```

```
im_stack = []

n_clip = 40

for nit in tqdm(iterations_list):

    im_deconvolved = restoration.richardson_lucy(im_noisy, psf, num_iter=nit, clip=

    im_deconvolved_clipped = im_deconvolved[n_clip:-n_clip, n_clip:-n_clip]

    im_deconvolved_clipped_rescaled = 255 * (im_deconvolved_clipped - np.min(im_dec
    im_deconvolved_clipped_rescaled = im_deconvolved_clipped_rescaled.astype(np.uin

    RMSE = calculate_image_RMSE(im[n_clip:-n_clip, n_clip:-n_clip], im_deconvolved_

    RMSE_list.append(RMSE)
    im_stack.append(im_deconvolved_clipped_rescaled)

plt.plot(iterations_list, RMSE_list)
plt.xlabel(r'$N_{iterations}$ of Deconvolution')
plt.ylabel(r'RMS Error');
```

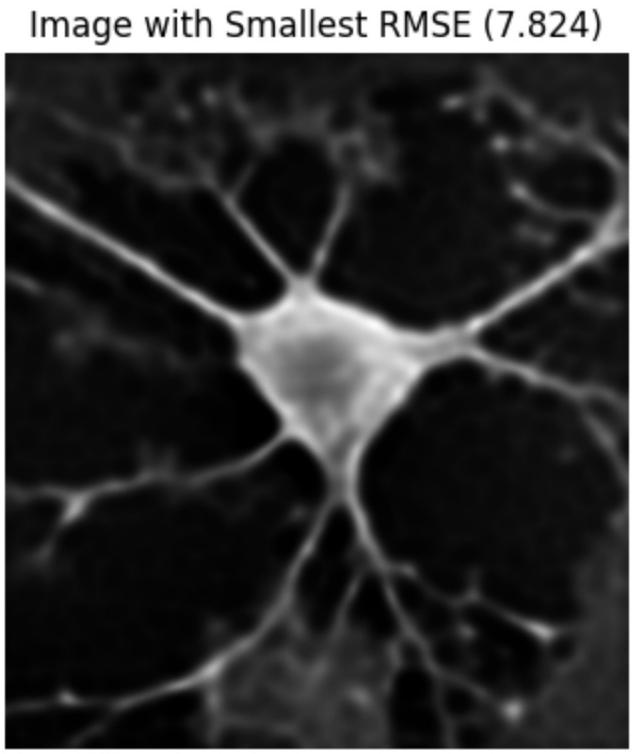
```
0%|          | 0/200 [00:00<?, ?it/s]
```



(e) Submit the most accurate deconvolved image (i.e. the image with the lowest RMSE)

In [119…

```
im_deconvolved_min_RMSE = im_stack[RMSE_list.index(min(RMSE_list))]
showim(im_deconvolved_min_RMSE, titles=f"Image with Smallest RMSE ({RMSE_list[RMSE_
```



(f)

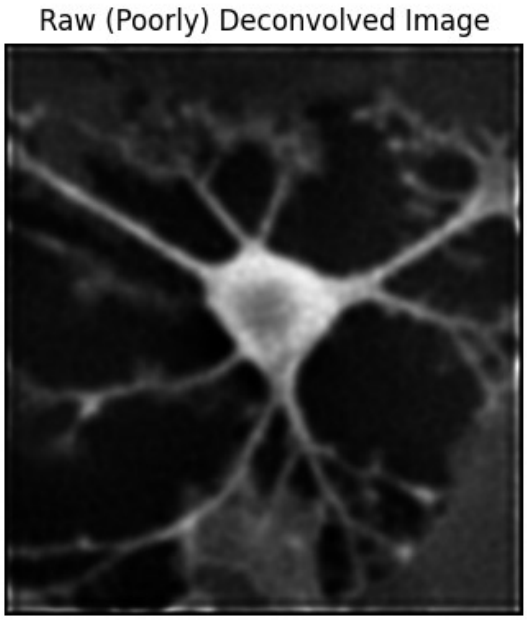
How does it look if we use the wrong PSF for deconvolution?

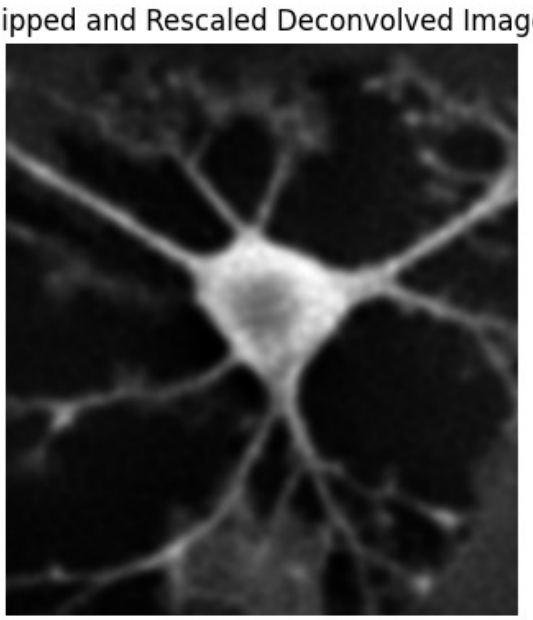
Repeat the above (c-e) using sigma = 5 for the deconvolution – i.e. we think the PSF is narrower than it actually is!

In [120…

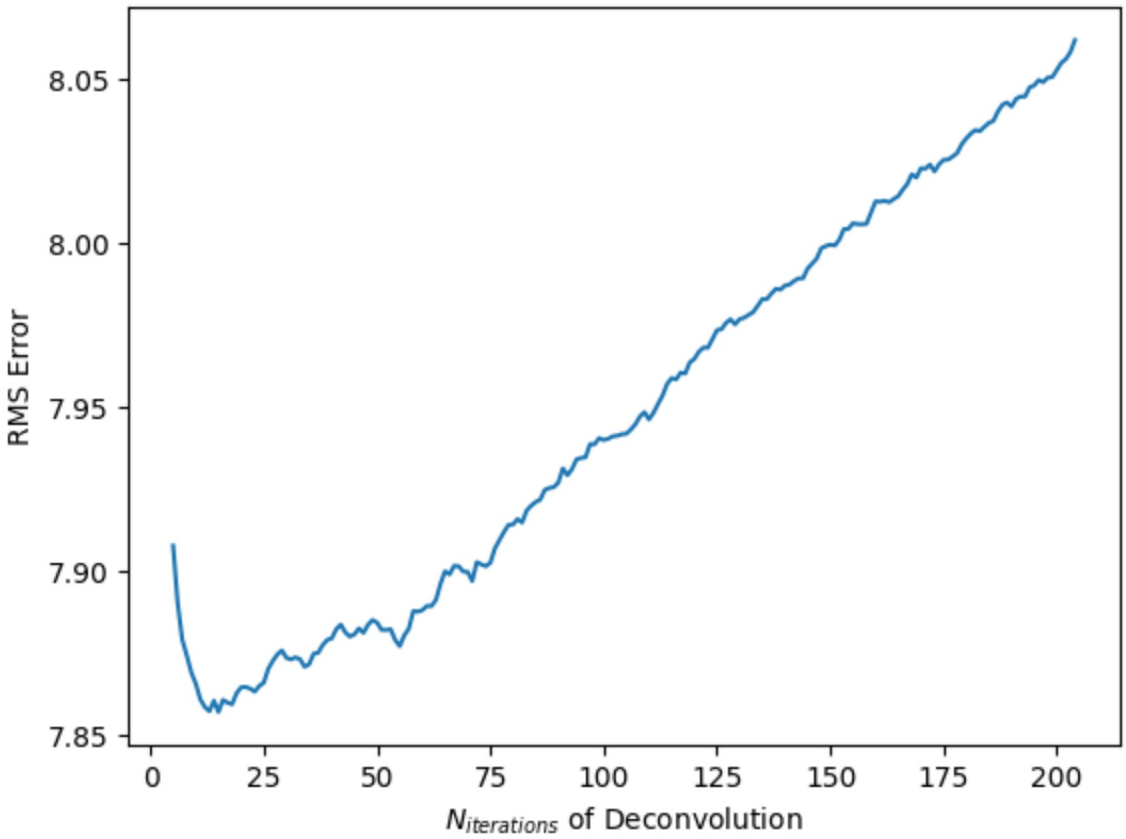
```
poor_psf = gaussian_psf(N, 5)

iterations = 20

im_deconvolved = restoration.richardson_lucy(im_noisy, poor_psf, num_iter=iteration

n_clip = 40 # I found 40 to be able to deal with the iterations = 205 case

im_deconvolved_clipped = im_deconvolved[n_clip:-n_clip, n_clip:-n_clip]
im_deconvolved_clipped_rescaled = 255 * (im_deconvolved_clipped - np.min(im_deconvo
im_deconvolved_clipped_rescaled = im_deconvolved_clipped_rescaled.astype(np.uint8)

RMSE_im_deconvolved = calculate_image_RMSE(im, im_deconvolved)
RMSE_im_deconvolved_clipped = calculate_image_RMSE(im[n_clip:-n_clip, n_clip:-n_cli

showim((im_deconvolved, im_deconvolved_clipped_rescaled), titles = ("Raw (Poorly) D

print(f"Raw deconvolved RMSE: {RMSE_im_deconvolved:.3f} || Clipped and rescaled dec

iterations_list_poor = np.arange(5,205,1)

RMSE_list_poor = []
im_stack_poor = []
```

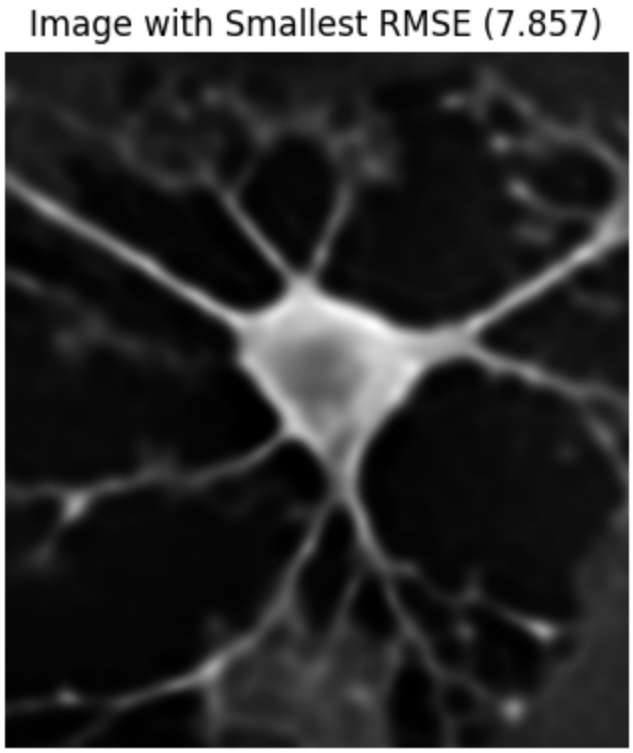
```
n_clip = 40

for nit in tqdm(iterations_list_poor):

    im_deconvolved = restoration.richardson_lucy(im_noisy, poor_psf, num_iter=nit,

    im_deconvolved_clipped = im_deconvolved[n_clip:-n_clip, n_clip:-n_clip]

    im_deconvolved_clipped_rescaled = 255 * (im_deconvolved_clipped - np.min(im_dec
    im_deconvolved_clipped_rescaled = im_deconvolved_clipped_rescaled.astype(np.uin

    RMSE = calculate_image_RMSE(im[n_clip:-n_clip, n_clip:-n_clip], im_deconvolved_

    RMSE_list_poor.append(RMSE)
    im_stack_poor.append(im_deconvolved_clipped_rescaled)

plt.plot(iterations_list_poor, RMSE_list_poor)
plt.xlabel(r'$N_{iterations}$ of Deconvolution')
plt.ylabel(r'RMS Error');

im_deconvolved_min_RMSE_poor = im_stack[RMSE_list_poor.index(min(RMSE_list_poor))]
showim(im_deconvolved_min_RMSE_poor, titles=f"Image with Smallest RMSE ({RMSE_list_
```





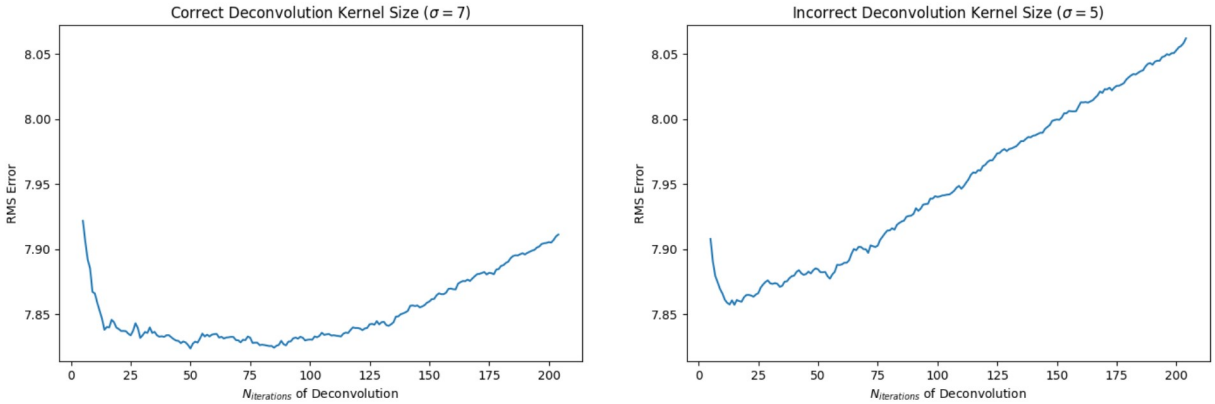

```
Raw deconvolved RMSE: 30.574 || Clipped and rescaled deconvolved RMSE: 7.865
  0%|          | 0/200 [00:00<?, ?it/s]
```





In [ ]:
```
plt.figure(figsize=(17,5))

ymax = max(RMSE_list_poor) + 0.01
ymin = min(RMSE_list) - 0.01

plt.subplot(1,2,1)
```

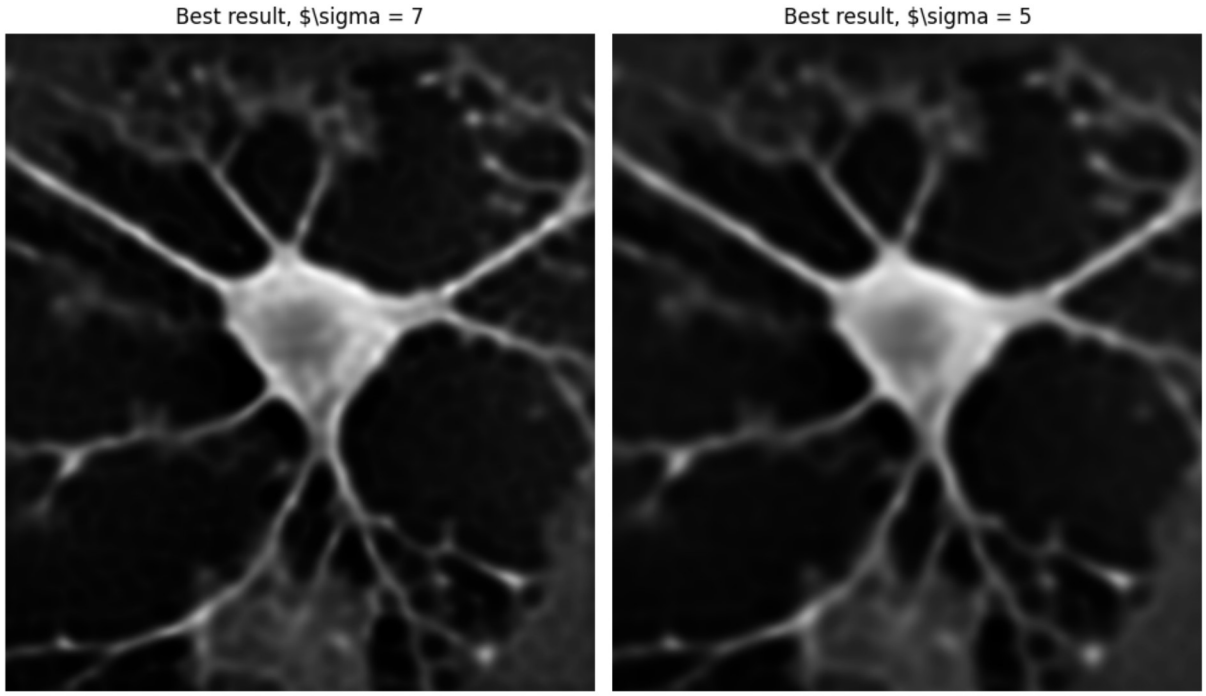
```
plt.title(r"Correct Deconvolution Kernel Size ($\sigma = 7$)")
plt.plot(iterations_list, RMSE_list)
plt.xlabel(r'$N_{iterations}$ of Deconvolution')
plt.ylabel(r'RMS Error')
plt.ylim([ymin, ymax])

plt.subplot(1,2,2)
plt.plot(iterations_list_poor, RMSE_list_poor)
plt.title(r"Incorrect Deconvolution Kernel Size ($\sigma = 5$)")
plt.xlabel(r'$N_{iterations}$ of Deconvolution')
plt.ylabel(r'RMS Error')
plt.ylim([ymin, ymax]);
```



With a narrower kernel size than the one used to create the convolved image, we get a slightly higher RMSE and quicker "takeoff" in error as iterations increase past the optimal.

In [141…

```
showim((im_deconvolved_min_RMSE, im_deconvolved_min_RMSE_poor), titles=(r"Best resu
```



Qualitatively, we see a "blurrier" deconvolved image when we use the wrong kernel size in deconvolution!