HW 5 Aiden Dillon

In [19]:
```
import numpy as np
import matplotlib.pyplot as plt
from scipy.special import j1
from tqdm.auto import tqdm
```

In [20]:
```
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 calculate_rms_error(calculated_positions, true_positions):
    """
    Calculate the RMS error between calculated and true centroid positions.

    Parameters:
    calculated_positions (list of tuples or numpy array): List or array of (x_c, y_
    true_positions (list of tuples or numpy array): List or array of (x_0, y_0) tru

    Returns:
    float: The RMS error in pixels.
    """

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

    # Ensure arrays are 2D, even if only one position is provided
    if calculated_positions.ndim == 1:
        calculated_positions = calculated_positions.reshape(1, -1)
    if true_positions.ndim == 1:
        true_positions = true_positions.reshape(1, -1)

    # Calculate squared differences
    squared_diffs = (calculated_positions - true_positions) ** 2

    # Sum the squared differences in x and y, then take the mean and square root
    rms_error = np.sqrt(np.mean(np.sum(squared_diffs, axis=1)))

    return rms_error


def calculate_centroid(intensities, camera_scale = None, origin_center=False):
    """
    Calculate the centroid (center of mass) for 1D or 2D intensity arrays,
    with an option to shift the origin to the center of the array.

    Parameters:
    intensities (np.array): Array of intensity values, either 1D or 2D.
    origin_center (bool): If True, shifts the origin to the center of the array.
                          If False, uses the bottom-left corner as the origin.

    Returns:
    tuple: The centroid position. For 1D, returns (x_centroid,).
           For 2D, returns (x_centroid, y_centroid).
```

```
    """
    if intensities.ndim == 1:
        # 1D case
        n = len(intensities)
        positions = np.arange(n)
        if origin_center:
            center = (n - 1) / 2
            positions = positions - center
        x_centroid = np.sum(positions * intensities) / np.sum(intensities)
        if camera_scale:
            return x_centroid*camera_scale
        else:
            return x_centroid

    elif intensities.ndim == 2:
        # 2D case
        m, n = intensities.shape
        x_positions = np.arange(n)
        y_positions = np.arange(m)

        if origin_center:
            x_center = (n - 1) / 2
            y_center = (m - 1) / 2
            x_positions = x_positions - x_center
            y_positions = y_positions - y_center

        x_weights = np.sum(intensities, axis=0)
        y_weights = np.sum(intensities, axis=1)

        x_centroid = np.sum(x_positions * x_weights) / np.sum(intensities)
        y_centroid = np.sum(y_positions * y_weights) / np.sum(intensities)

        if camera_scale:
            return (x_centroid*camera_scale, y_centroid*camera_scale)
        else:
            return (x_centroid, y_centroid)

    else:
        raise ValueError("Only 1D and 2D arrays are supported")


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
```

1) SNR Ratio in Images

Part (i): Math Explanation

WTS: multiplying all X values by a constant A does not change the Signal-to-Noise Ratio (SNR).

The SNR is defined as: $\mathrm{SNR} = \frac{\langle x \rangle}{\sigma}$ where $\langle x \rangle$ is the mean of (x) and $(sigma)$ is the standard deviation of (x).

If we multiply each (x) value by a constant (A). Then, the new values become $(A \cdot x)$

The mean of (A * x) is: $\langle A \cdot x \rangle = A \cdot \langle x \rangle$

The standard deviation of (A * x) is:

$$\sigma_{A \cdot x} = \sqrt{\langle (A \cdot x - \langle A \cdot x \rangle)^2 \rangle} = A \cdot \sigma$$

Therefore, the SNR for (A * x) is:

$$\mathrm{SNR}_{A \cdot x} = \frac{\langle A \cdot x \rangle}{\sigma_{A \cdot x}} = \frac{A \cdot \langle x \rangle}{A \cdot \sigma} = \frac{\langle x \rangle}{\sigma} = \mathrm{SNR}$$

Thus, multiplying all X values by a constant A does not change the SNR.

Problem 2: *Centroid Warmup*

(i)

Consider intensities I_i = 0.2 i^0.5, where the position index i goes from 0 to 999. (In other words, the intensity increases as the square root of position). Where is the centroid?

In [21]:
```
positions = np.arange(1000)

intensities = 0.2 * np.sqrt(positions)
centroid = calculate_centroid(intensities)
print(f"Centroid position: {centroid:.2f}")
```

```
Centroid position: 599.71
```

(ii)

In [22]:
```
w = np.random.poisson(10, size=1000)
intensities = 30 / (positions + 20) + 0.05 * w
centroid = calculate_centroid(intensities)
print(f"Centroid position: {centroid:.2f}")
```

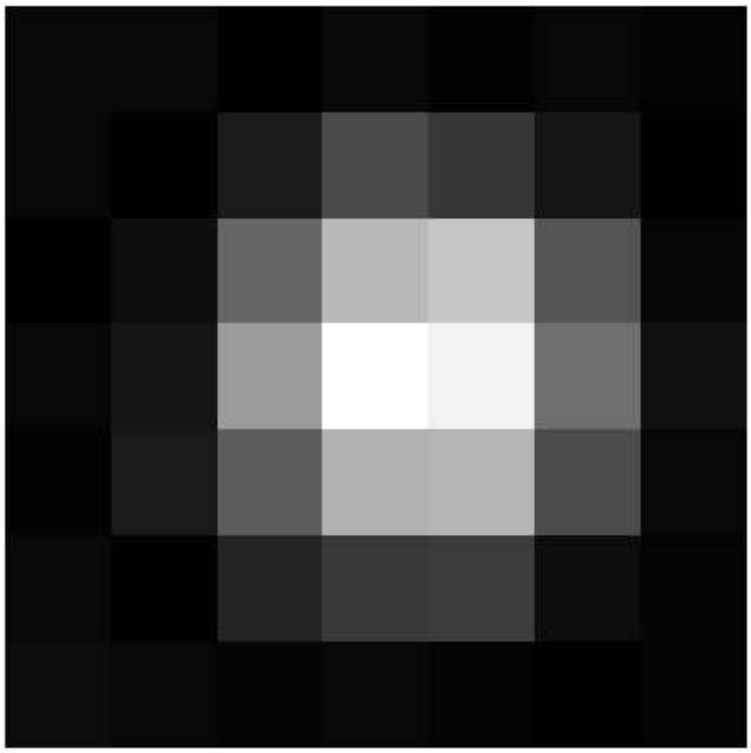
```
Centroid position: 445.58
```

Problem 3: *Centroid Localization*

The below section is just running code to check my simuluated images

In [23]:
```
def checkImage(simImage):

    N = simImage.shape[0] # won't check that array is square
    center_px = int((N-1)/2)
    ratio_center_right1px = simImage[center_px,center_px] / simImage[center_px,cent
    print(f'Ratio #1: {ratio_center_right1px:.2f}')
    if np.abs(ratio_center_right1px-1.06)<0.18:
        print('Test 1: good')
    else:
        print('Test 1: failed!')
    ratio_center_left1px = simImage[center_px,center_px] / simImage[center_px,cente
    print(f'Ratio #2: {ratio_center_left1px:.2f}')
    if np.abs(ratio_center_left1px-1.81)<0.4:
        print('Test 2: good')
    else:
        print('Test 2: failed!')
    ratio_center_up1px = simImage[center_px,center_px] / simImage[center_px-1,cente
    print(f'Ratio #3: {ratio_center_up1px:.2f}')
    if np.abs(ratio_center_up1px-1.36)<0.26:
        print('Test 3: good')
    else:
        print('Test 3: failed!')

N_camera = 7
wavelength = 0.50
NA = 0.9
```

```
camera_scale = 0.1
fine_scale = 0.01
N_photons = 2000
center = (0.04, 0)
bg = 2

test = sim_ps(N_camera=N_camera, wavelength=wavelength, NA=NA, camera_scale=camera_
              fine_scale=fine_scale, N_photon=N_photons, center = center, B=bg)


showim(test)

checkImage(test)
```



```
Ratio #1: 1.06
Test 1: good
Ratio #2: 1.65
Test 2: good
Ratio #3: 1.38
Test 3: good
```

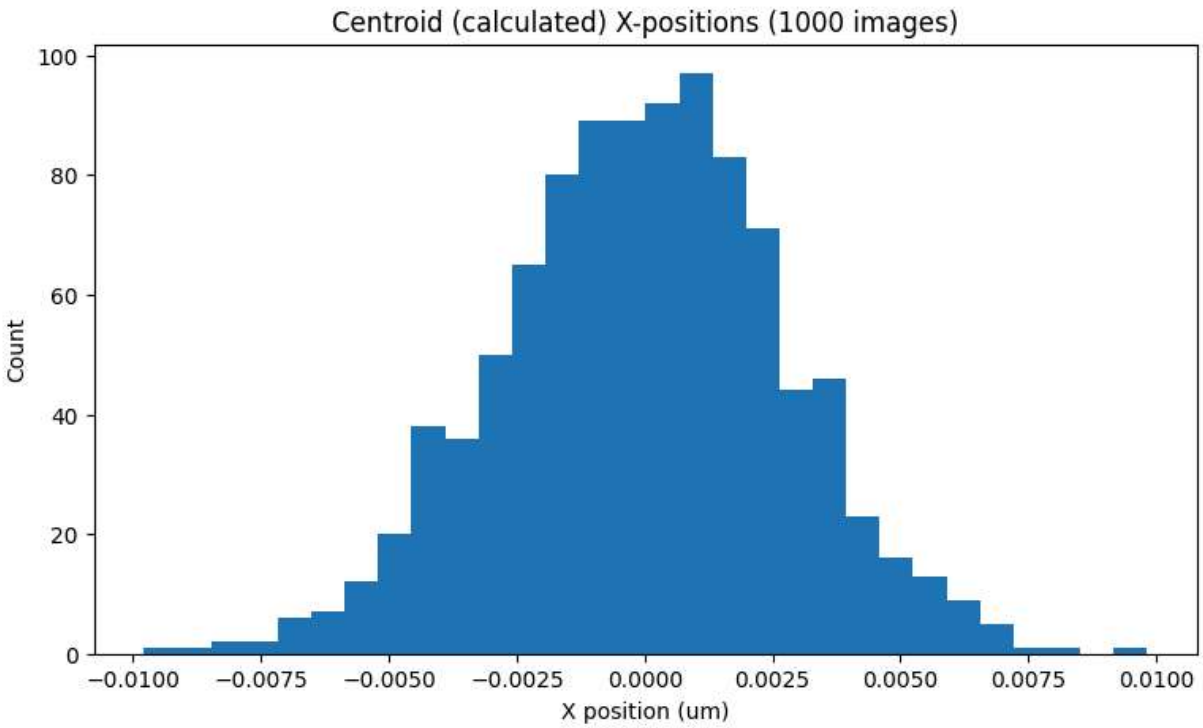
(a)

Calculating RMS of centroid for many simulated point source images

In [24]:
```
N_camera = 7
wavelength = 0.510
NA = 0.9
camera_scale = 0.1
fine_scale = 0.001
N_photons = 2000
center = (0.0, 0.0)
bg = 10
```

```
centers = []
centroids = []
x_list = []

m = 1000

for _ in range(m):

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

    centroid = calculate_centroid(image, origin_center=True, camera_scale=camera_sc
    center = center

    centroids.append(centroid)
    centers.append(center)
    x_list.append(centroid[0])

rms = calculate_rms_error(centers, centroids)
print(f"Centroid root mean squared error for {m} images: {rms:5f} um")

plt.figure(figsize=(9,5))
plt.hist(x_list, bins=30)
plt.title(f"Centroid (calculated) X-positions ({m} images)")
plt.xlabel('X position (um)')
plt.ylabel('Count');
```

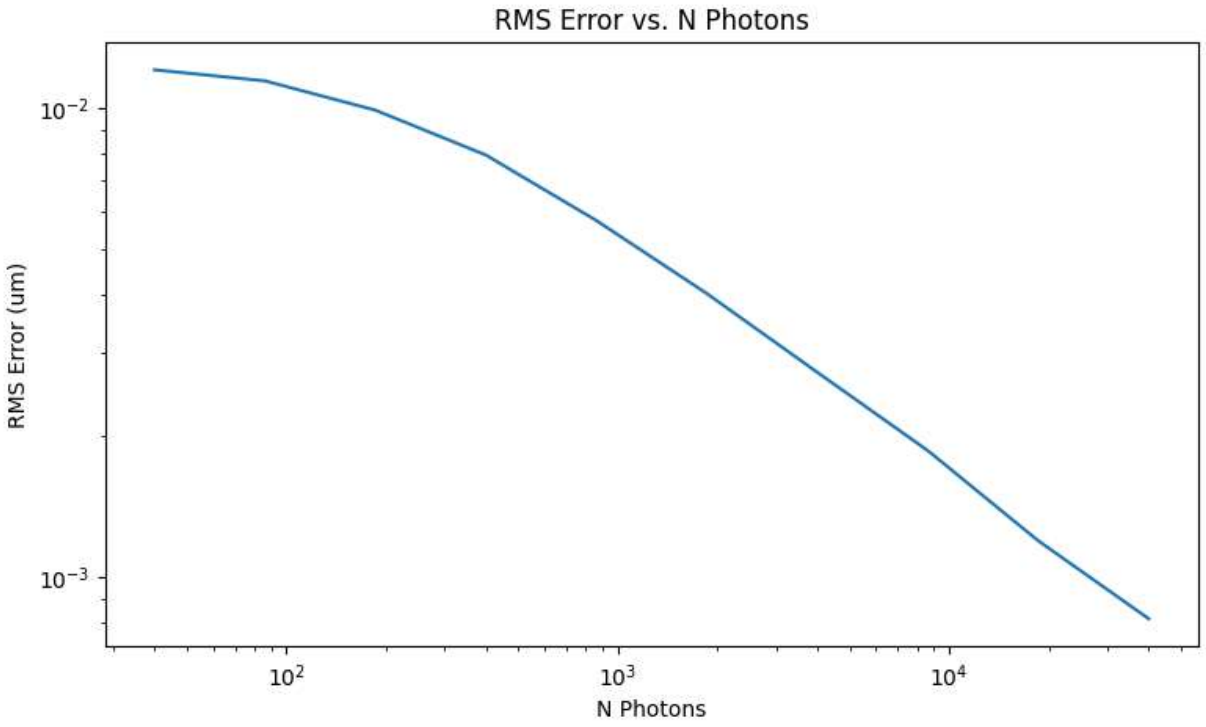
```
Centroid root mean squared error for 1000 images: 0.003865 um
```



(b)

RMS Error as N_Photons changes

In [25]:
```
N_camera = 7
wavelength = 0.510
NA = 0.9
camera_scale = 0.1
fine_scale = 0.001
center = (0.0, 0.0)
bg = 10

m = 1000
rms_list = []
N_photons_list = np.logspace(np.log10(40), np.log10(40000), num=10)

for N_photons in N_photons_list:

    centers = []
    centroids = []

    N_photons = N_photons

    for _ in range(m):

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

        centroid = calculate_centroid(image, origin_center=True, camera_scale=camer
        center = center

        centroids.append(centroid)
        centers.append(center)

    rms = calculate_rms_error(centers, centroids)
    rms_list.append(rms)

plt.figure(figsize=(9,5))
plt.loglog(N_photons_list,rms_list)
plt.xlabel('N Photons')
plt.ylabel('RMS Error (um)')
plt.title('RMS Error vs. N Photons');
```



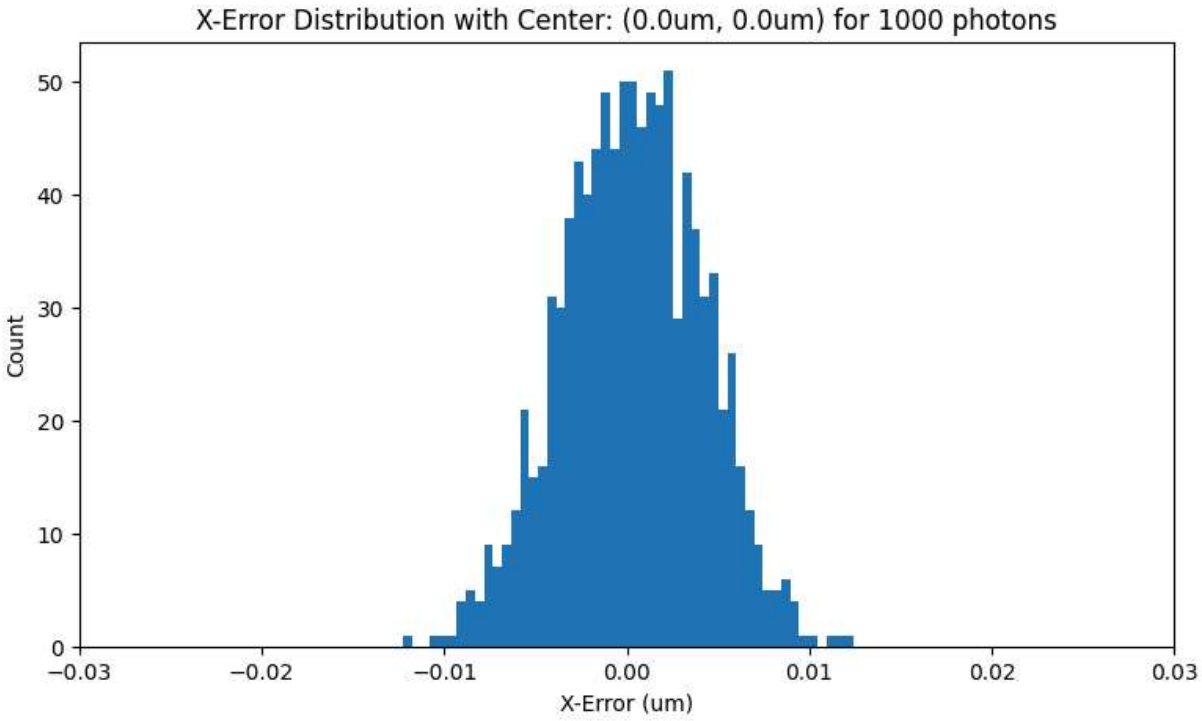
The Shape of the graph makes sense, as we expect the RMS error to go down as we increase the amount of photons. Since SNR is proportional to sqrt(N_photon), and SNR and RMS error are inversely proportional, we expect a relationship of RMS proportional to 1/sqrt(N_photons) which is reflected in the graph above (after an initial kickoff period of small N photons).

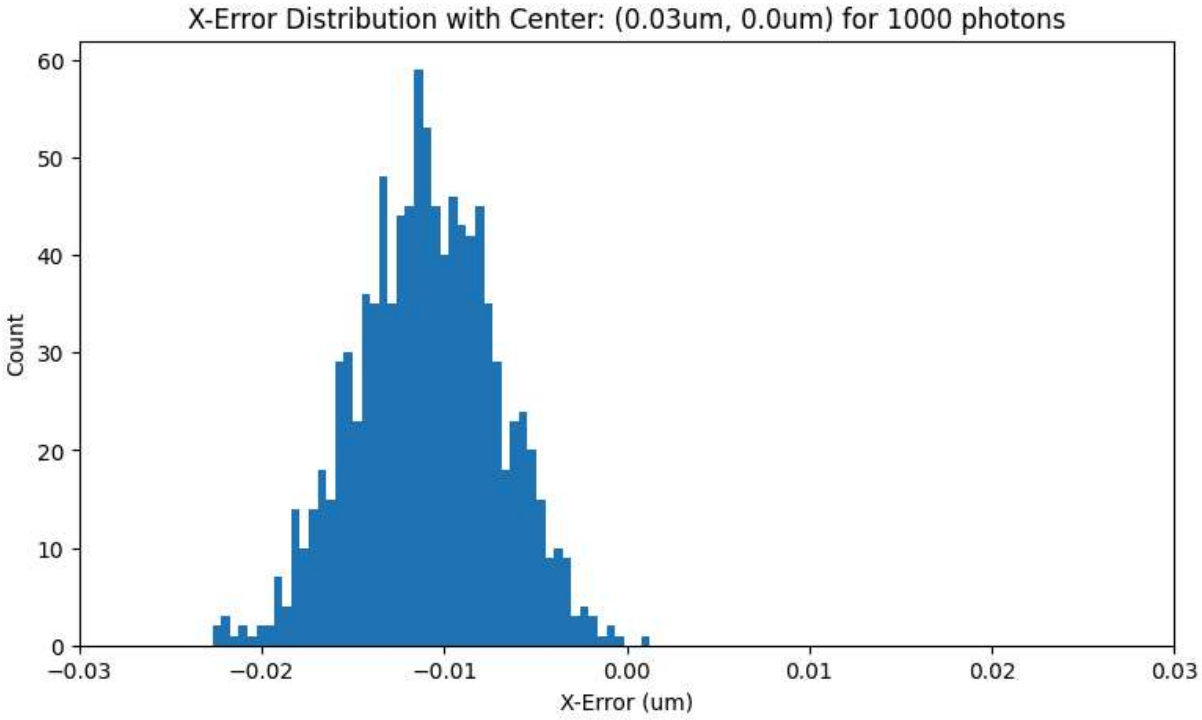
(c)

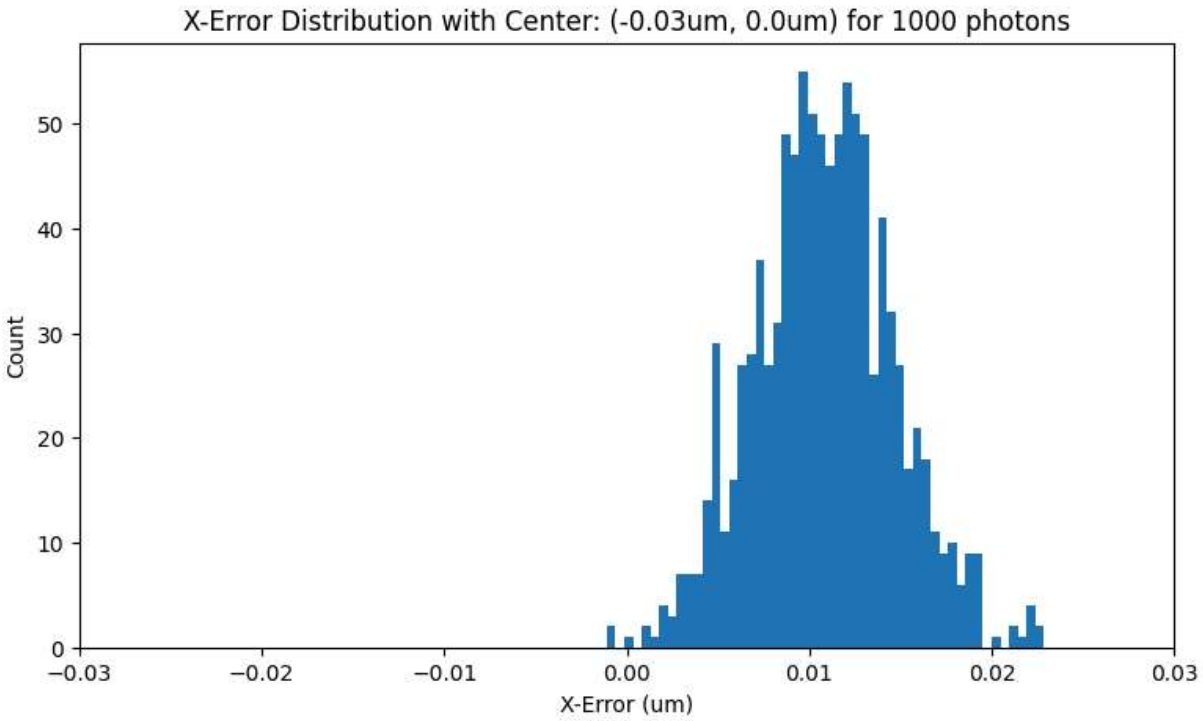
X-error as simulated point source center changes

In [26]:
```
N_camera = 7
wavelength = 0.510
NA = 0.9
N_photons = 1000
camera_scale = 0.1
fine_scale = 0.001
bg = 10

m = 1000
center_list = [(0.0, 0.0), (0.03, 0.0), (-0.03, 0.0)]

for center in center_list:

    x_error_list = []

    for _ in range(m):

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

        centroid = calculate_centroid(image, origin_center=True, camera_scale=camer
```

```
        x_centroid = centroid[0]
        x_error = x_centroid - center[0]

        x_error_list.append(x_error)

    plt.figure(figsize=(9,5))
    plt.hist(x_error_list, bins=50)
    plt.xlim(-0.03, 0.03)
    plt.xlabel('X-Error (um)')
    plt.ylabel('Count')
    plt.title(f"X-Error Distribution with Center: ({center[0]}um, {center[1]}um) fo
```



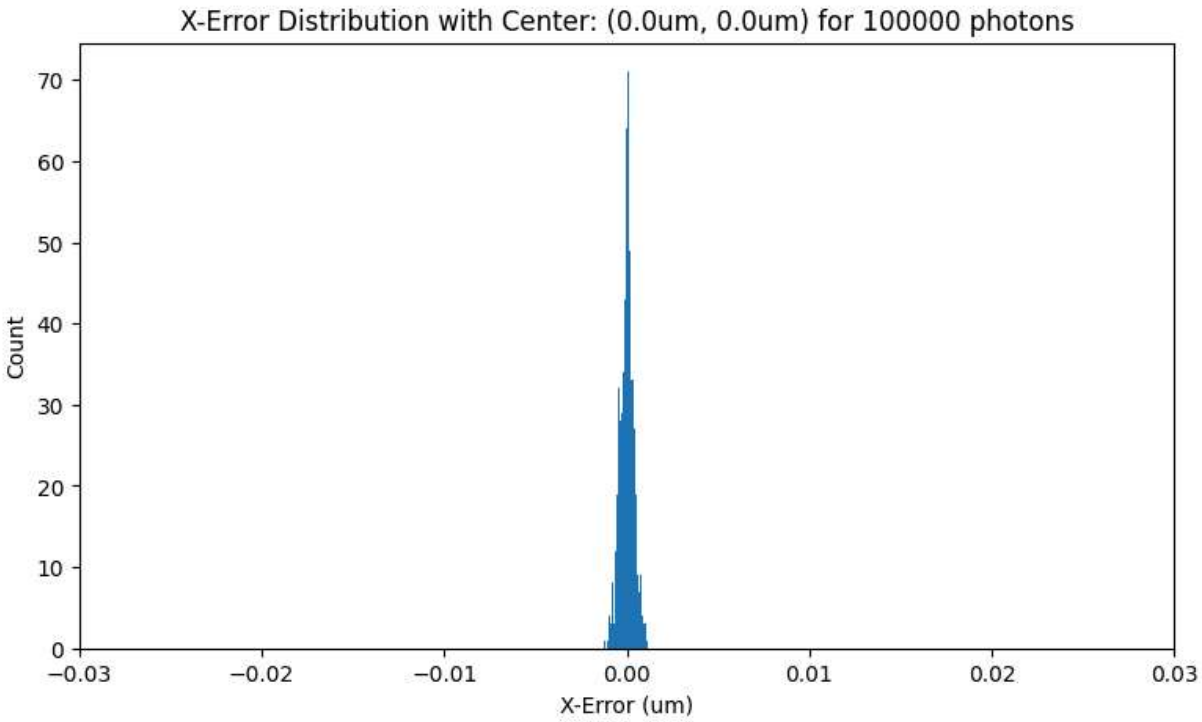


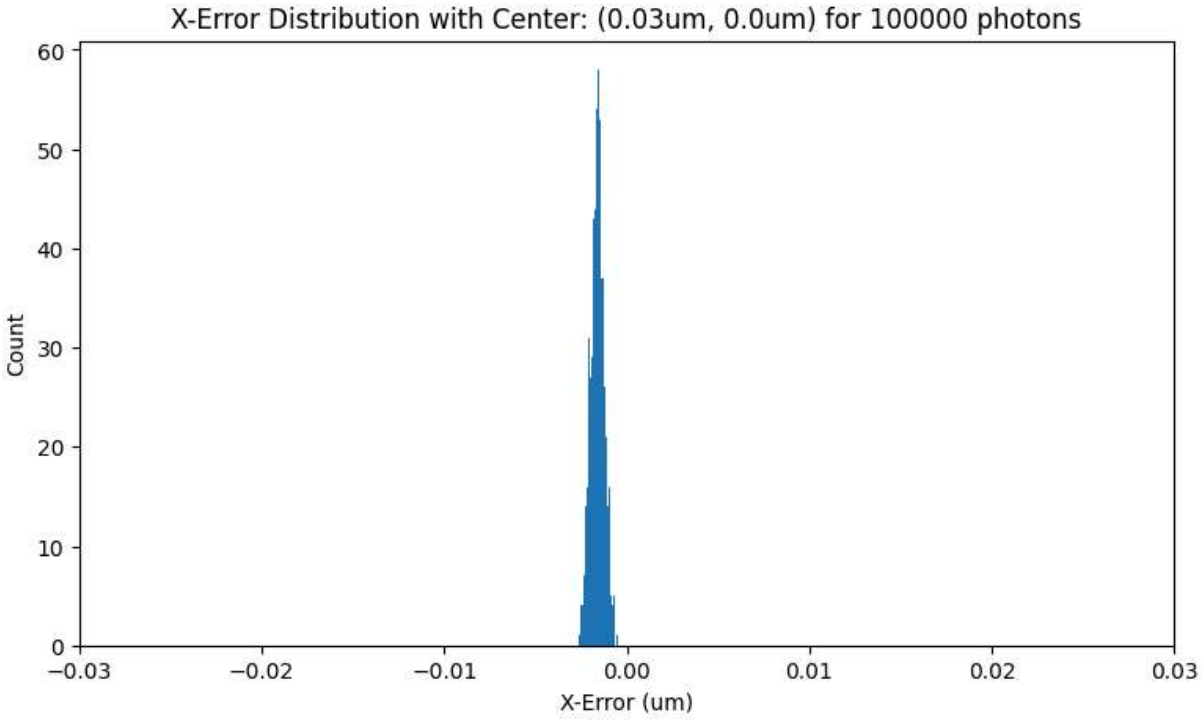


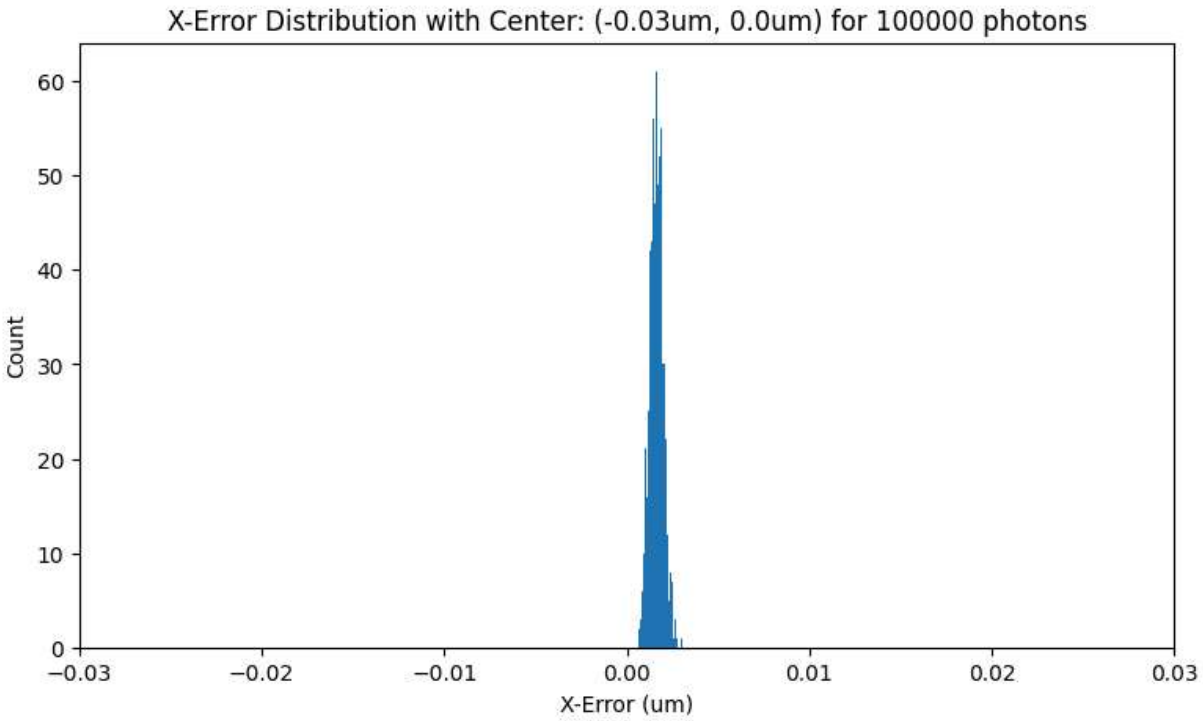
The centroid is not an unbiased estimator, as the error is dependent on the true value. The error seems to be normally distributed, but shifts its value as the center shifts

In [27]:
```
N_camera = 7
wavelength = 0.510
NA = 0.9
N_photons = 100000
camera_scale = 0.1
fine_scale = 0.001
bg = 10

m = 1000
center_list = [(0.0, 0.0), (0.03, 0.0), (-0.03, 0.0)]

for center in center_list:

    x_error_list = []

    for _ in range(m):

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

        centroid = calculate_centroid(image, origin_center=True, camera_scale=camer
        x_centroid = centroid[0]
        x_error = x_centroid - center[0]

        x_error_list.append(x_error)

    plt.figure(figsize=(9,5))
    plt.hist(x_error_list, bins=50)
    plt.xlim(-0.03, 0.03)
    plt.xlabel('X-Error (um)')
```

```
    plt.ylabel('Count')
    plt.title(f"X-Error Distribution with Center: ({center[0]}um, {center[1]}um) fo
```





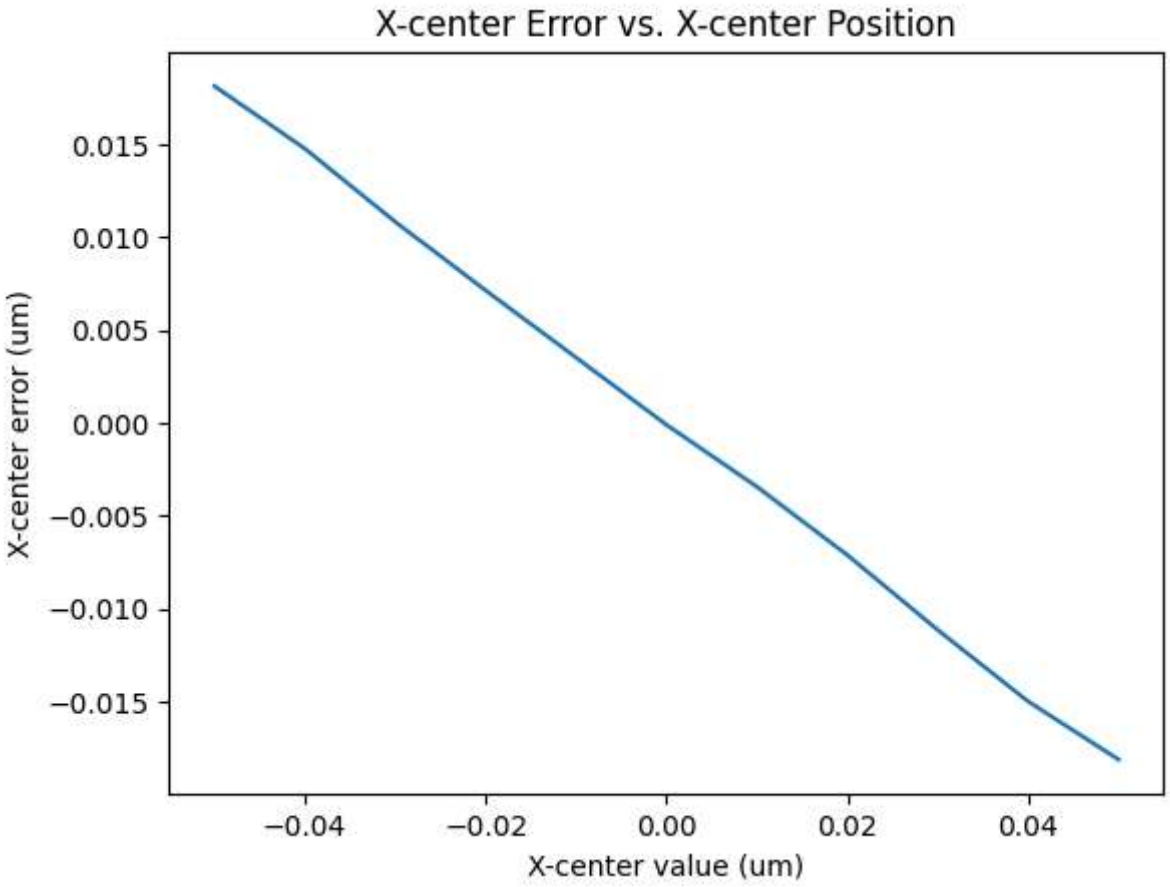


While the distribution of error is now tighter, and it "moves" less as the center moves, it is still a biased estimator. It seems that increasing N_photons reduces the effects, but they are still present.

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

center_x_list = np.linspace(-0.05, 0.05, 11)

m = 100

mean_x_error_list = []

for center_x in center_x_list:

    center = (center_x, 0.0)

    x_error_list = []

    for _ in range(m):

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

        centroid = calculate_centroid(image, origin_center=True, camera_scale=camer
        x_centroid = centroid[0]
        x_error = x_centroid - center[0]
```

```
        x_error_list.append(x_error)

    mean_x_error = np.mean(x_error_list)
    mean_x_error_list.append(mean_x_error)

plt.plot(center_x_list, mean_x_error_list)
plt.xlabel('X-center value (um)')
plt.ylabel('X-center error (um)')
plt.title('X-center Error vs. X-center Position')
```

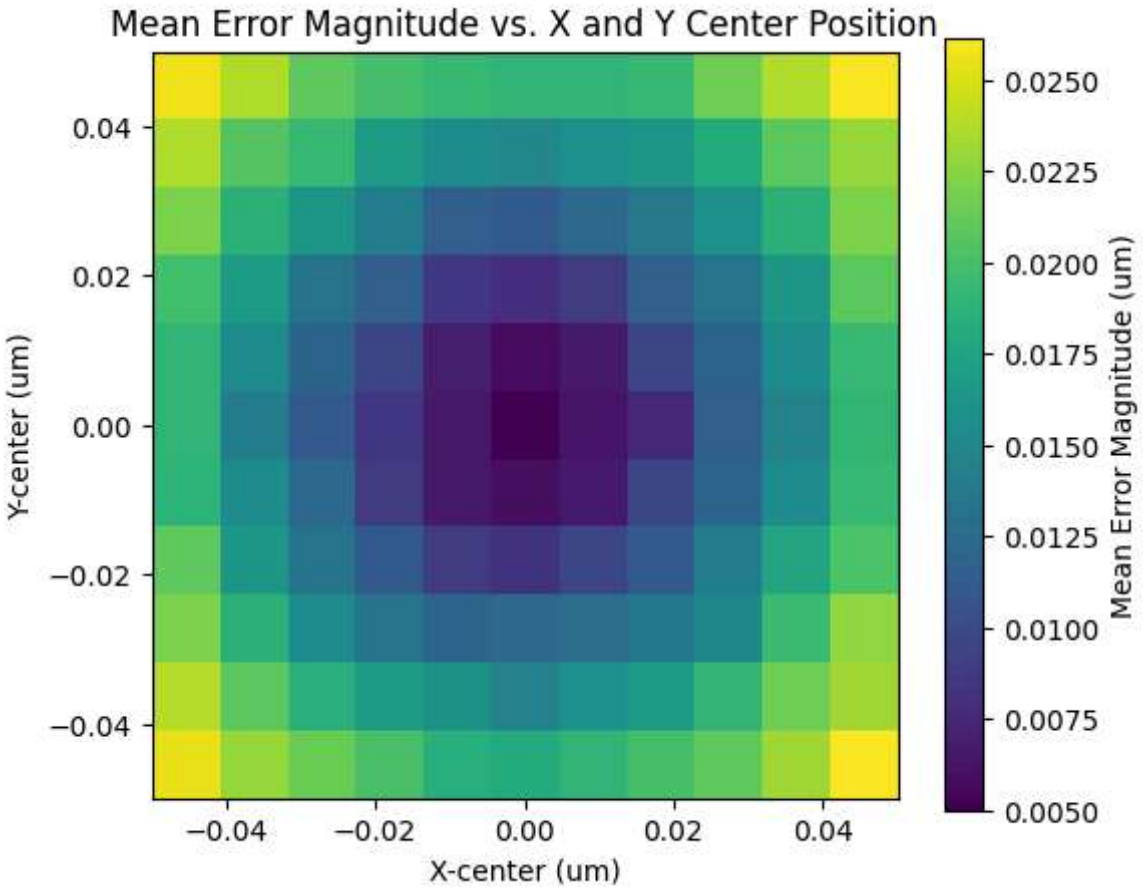
Out[28]: Text(0.5, 1.0, 'X-center Error vs. X-center Position')



In [29]:
```
N_camera = 7
wavelength = 0.510
NA = 0.9
N_photons = 1000
camera_scale = 0.1
fine_scale = 0.001
bg = 10
m = 100

center_x_list = np.linspace(-0.05, 0.05, 11)
center_y_list = np.linspace(-0.05, 0.05, 11)

mean_error_magnitude_array = np.zeros((len(center_x_list), len(center_y_list)))

for i, center_x in tqdm(enumerate(center_x_list), total = len(center_x_list), desc
    for j, center_y in tqdm(enumerate(center_y_list), total=len(center_y_list),
```

```
                            desc=f"Processing Y positions for {center_x:.4f}", lea

        center = (center_x, center_y)

        error_magnitude_list = []

        for _ in range(m):
            image = sim_ps(N_camera=N_camera, wavelength=wavelength, NA=NA, camera_
                           fine_scale=fine_scale, N_photon=N_photons, center=center

            centroid = calculate_centroid(image, origin_center=True, camera_scale=c
            x_centroid, y_centroid = centroid

            error_magnitude = np.sqrt((x_centroid - center_x) ** 2 + (y_centroid -
            error_magnitude_list.append(error_magnitude)

        mean_error_magnitude_array[i, j] = np.mean(error_magnitude_list)

plt.figure(figsize=(6, 5))
c = plt.imshow(mean_error_magnitude_array, extent=[center_x_list[0], center_x_list[
               origin='lower', cmap='viridis')
plt.colorbar(c, label="Mean Error Magnitude (um)")
plt.title("Mean Error Magnitude vs. X and Y Center Position")
plt.xlabel("X-center (um)")
plt.ylabel("Y-center (um)")
plt.show()
```

```
Outer Loop, X Center = 0.0500 :   0%|          | 0/11 [00:00<?, ?it/s]
Processing Y positions for -0.0500:   0%|          | 0/11 [00:00<?, ?it/s]
Processing Y positions for -0.0400:   0%|          | 0/11 [00:00<?, ?it/s]
Processing Y positions for -0.0300:   0%|          | 0/11 [00:00<?, ?it/s]
Processing Y positions for -0.0200:   0%|          | 0/11 [00:00<?, ?it/s]
Processing Y positions for -0.0100:   0%|          | 0/11 [00:00<?, ?it/s]
Processing Y positions for 0.0000:   0%|          | 0/11 [00:00<?, ?it/s]
Processing Y positions for 0.0100:   0%|          | 0/11 [00:00<?, ?it/s]
Processing Y positions for 0.0200:   0%|          | 0/11 [00:00<?, ?it/s]
Processing Y positions for 0.0300:   0%|          | 0/11 [00:00<?, ?it/s]
Processing Y positions for 0.0400:   0%|          | 0/11 [00:00<?, ?it/s]
Processing Y positions for 0.0500:   0%|          | 0/11 [00:00<?, ?it/s]
```

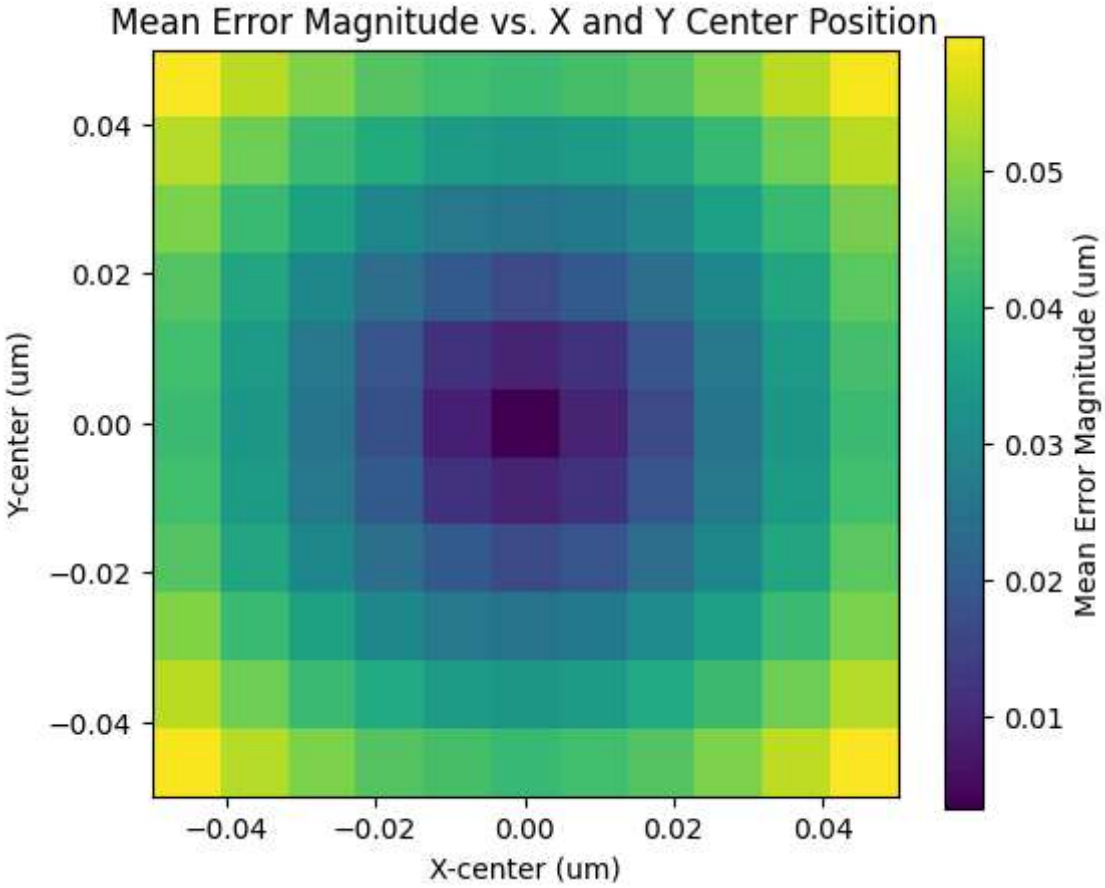


There is a clear bias, with error being the least when p is ~0, or when the center of the point source is closer to the center. Error increases as the point source moves towards the edges of the image.

In [30]:
```
N_camera = 7
wavelength = 0.510
NA = 0.9
N_photons = 1000
camera_scale = 0.1
fine_scale = 0.001
bg = 100
m = 100

center_x_list = np.linspace(-0.05, 0.05, 11)
center_y_list = np.linspace(-0.05, 0.05, 11)

mean_error_magnitude_array = np.zeros((len(center_x_list), len(center_y_list)))

for i, center_x in tqdm(enumerate(center_x_list), total = len(center_x_list), desc
    for j, center_y in tqdm(enumerate(center_y_list), total=len(center_y_list),
                            desc=f"Processing Y positions for {center_x:.4f}", lea

        center = (center_x, center_y)

        error_magnitude_list = []

        for _ in range(m):
```

```
            image = sim_ps(N_camera=N_camera, wavelength=wavelength, NA=NA, camera_
                           fine_scale=fine_scale, N_photon=N_photons, center=center

            centroid = calculate_centroid(image, origin_center=True, camera_scale=c
            x_centroid, y_centroid = centroid

            error_magnitude = np.sqrt((x_centroid - center_x) ** 2 + (y_centroid -
            error_magnitude_list.append(error_magnitude)

        mean_error_magnitude_array[i, j] = np.mean(error_magnitude_list)

plt.figure(figsize=(6, 5))
c = plt.imshow(mean_error_magnitude_array, extent=[center_x_list[0], center_x_list[
               origin='lower', cmap='viridis')
plt.colorbar(c, label="Mean Error Magnitude (um)")
plt.title("Mean Error Magnitude vs. X and Y Center Position")
plt.xlabel("X-center (um)")
plt.ylabel("Y-center (um)")
plt.show()
```

```
Outer Loop, X Center = 0.0500 :   0%|          | 0/11 [00:00<?, ?it/s]
Processing Y positions for -0.0500:   0%|          | 0/11 [00:00<?, ?it/s]
Processing Y positions for -0.0400:   0%|          | 0/11 [00:00<?, ?it/s]
Processing Y positions for -0.0300:   0%|          | 0/11 [00:00<?, ?it/s]
Processing Y positions for -0.0200:   0%|          | 0/11 [00:00<?, ?it/s]
Processing Y positions for -0.0100:   0%|          | 0/11 [00:00<?, ?it/s]
Processing Y positions for 0.0000:   0%|          | 0/11 [00:00<?, ?it/s]
Processing Y positions for 0.0100:   0%|          | 0/11 [00:00<?, ?it/s]
Processing Y positions for 0.0200:   0%|          | 0/11 [00:00<?, ?it/s]
Processing Y positions for 0.0300:   0%|          | 0/11 [00:00<?, ?it/s]
Processing Y positions for 0.0400:   0%|          | 0/11 [00:00<?, ?it/s]
Processing Y positions for 0.0500:   0%|          | 0/11 [00:00<?, ?it/s]
```



The error gets even worse as there is more background noise, (which is somewhat expected), indicating that the issue is that the centroid calculation has trouble dealing with the background noise as the center moves, well... off center

For the first topic of **image segmentation**, I read through some of *A Survey on Image Segmentation Techniques* by Sezgin and Sankur (2021), which provided a comprehensive overview of various methods, (both classical and machine-learning-based). Image segmentation divides an image into distinct regions to simplify analysis or extract meaningful objects. Image segmentation might be categorized into 3 types on the basis of the image properties: edge-based segmentation approaches, region-based segmentation approaches and thresholding segmentation approaches, along with deep neural network approaches, hybrid approaches and clustering approaches. Beyond traditional watershed segmentation, more advanced techniques include graph-based segmentation, where pixels are connected by edges representing similarity, and active contour models, also known as "snakes," which dynamically adjust to outline objects in an image. Some modern techniques incorporate machine learning, like U-Net, a convolutional network specifically designed for segmentation, and Mask R-CNN, which performs instance segmentation at the pixel level. One aspect I found particularly interesting is how segmentation techniques now often blend classical methods with machine learning. For example, graph-based segmentation can be enhanced by using learned edge weights to define boundaries more adaptively. Mask R-CNN's ability to detect and segment multiple instances within complex images showcases how ML has extended traditional segmentation. I really like the idea of combining more

robust techniques with some ML or deep learning to allow fine tuning/enhance traditional approaches. However, I found certain areas challenging, especially in understanding how similarity metrics are calculated for different color spaces and texture features. I'm also uncertain about how ML techniques would actually be applied in combination/parallel with traditional methods, since I'm used to only using one at a time.

For the second topic of **denoising** I read *Image Denoising Review: From Classical to State-of-the-Art Approaches* by Goyal et al. (2020), which outlines a range of de-noising techniques, from traditional methods to more advanced/adaptive approaches. The goal of de-noising is to reduce noise for a clearer, more accurate image. While simple techniques like low-pass filtering are common, they usually blur important details. Advanced methods, such as wavelet-based de-noising, work by transforming the image into wavelet coefficients, selectively reducing noise at various frequency levels and preserving finer details. Non-local means de-noising goes a step further, averaging pixel values based on neighborhoods with similar patterns across the image, which helps retain texture and edges without losing critical details. Something I found particularly interesting was how modern techniques focus on adaptability, adjusting noise reduction in different areas of the image to better preserve structure. This approach is especially evident in non-local means, which looks at similar patches across the entire image instead of relying solely on local information. One thing I'm unsure of/I'm sure will just take more experience is *when* to use which of these techniques, in reference to different types of noise.