Assessment for thesis

October 16, 2024

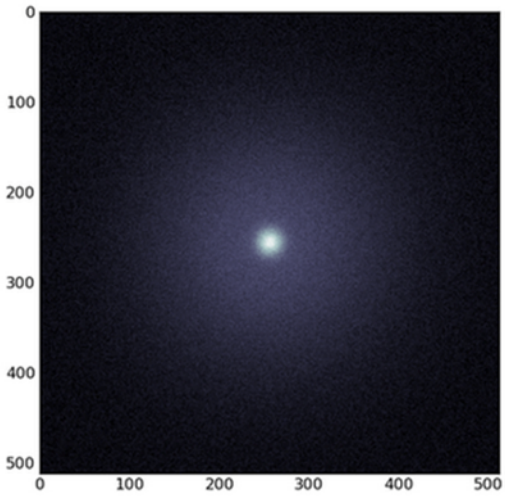
1 Python Assessment

[2]:
```
#import modules needed for the assessment
import numpy as np
import matplotlib.pyplot as plt
import random
from scipy.optimize import curve_fit
```

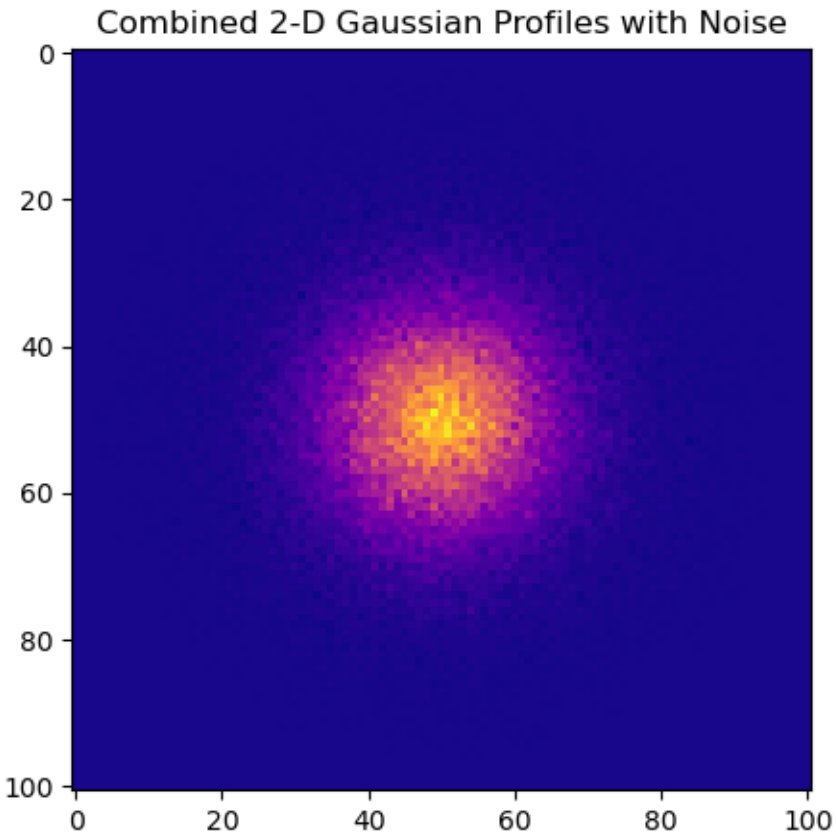
1.0.1 (a)

Generate a 101 by 101 zeros array, then add: * a flat "background": where each pixel value is drawn from a Normal distribution with a peak value of 5 and a sigma of 3. * a 2-D Gaussian profile with a max value of 100 and centered on location (50,50) and with a sigma of 10 pixels. Each pixel should have a value added to it that is Normal distribution about the value expected from the Gaussian model (and a sigma given by the square root of that value). * another 2-D Gaussian profile and with a max value of 20 centered on location (50,50) and with a sigma of 3 pixels. Each pixel should have a value added to it that is Normal distribution about the value expected from the [2nd] Gaussian model (and a sigma given by the square root of that value).

Your image should look like this (colour scale not important)



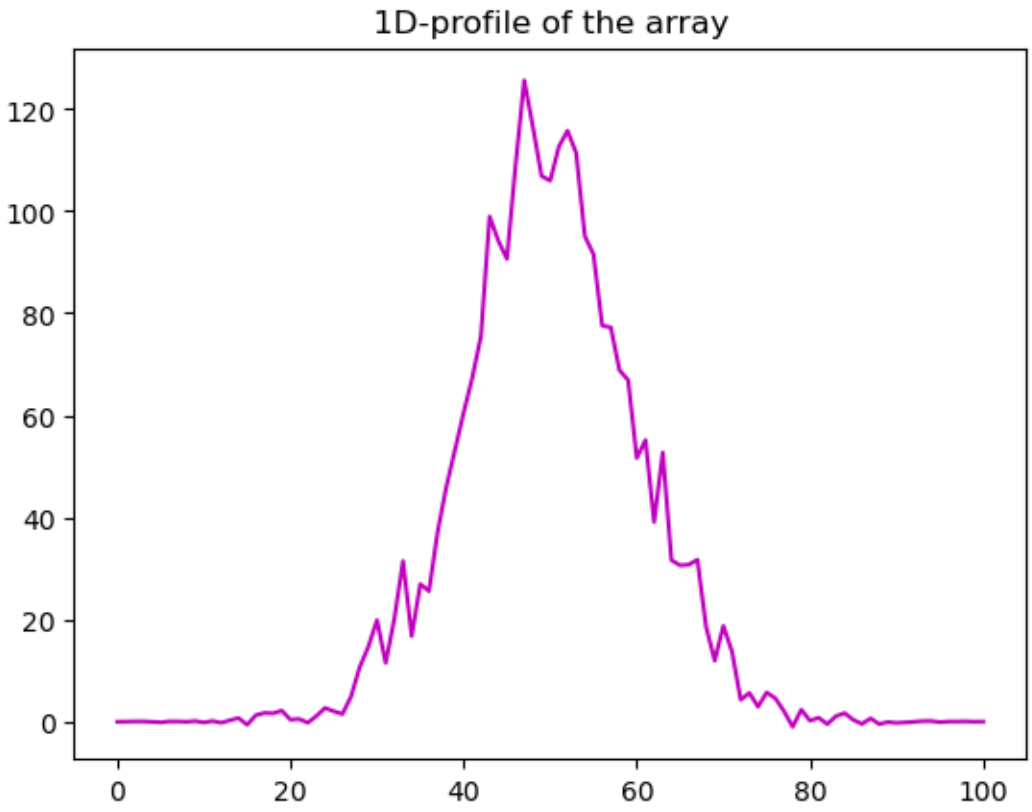
[4]:
```
#create the array with background noise
zero_array= np.zeros((101,101))
noise= np.random.normal(5, 3, zero_array.shape)

#add the 2D Gaussians with noise to the data
def gaussian(zero_array, max_value, center, sigma):
    x= np.linspace(0, zero_array.shape[0]-1, zero_array.shape[0])
    y= np.linspace(0, zero_array.shape[1]-1, zero_array.shape[1])
    x, y= np.meshgrid(x, y)
    gaussian_profile = max_value*np.exp(-((x-center[0])**2+(y-center[1])**2)/
 ↪(2*sigma**2))
    noise= np.random.normal(loc=0, scale=np.sqrt(gaussian_profile), size=␣
 ↪zero_array.shape)
    zero_array+= gaussian_profile + noise
    return zero_array

zero_array= gaussian(zero_array, max_value=100, center=(50, 50), sigma=10)
zero_array= gaussian(zero_array, max_value=20, center=(50, 50), sigma=3)

# Plot the result
plt.figure(figsize=(5, 5))
plt.imshow(zero_array, cmap='plasma')
plt.title('Combined 2-D Gaussian Profiles with Noise')
plt.show()
```

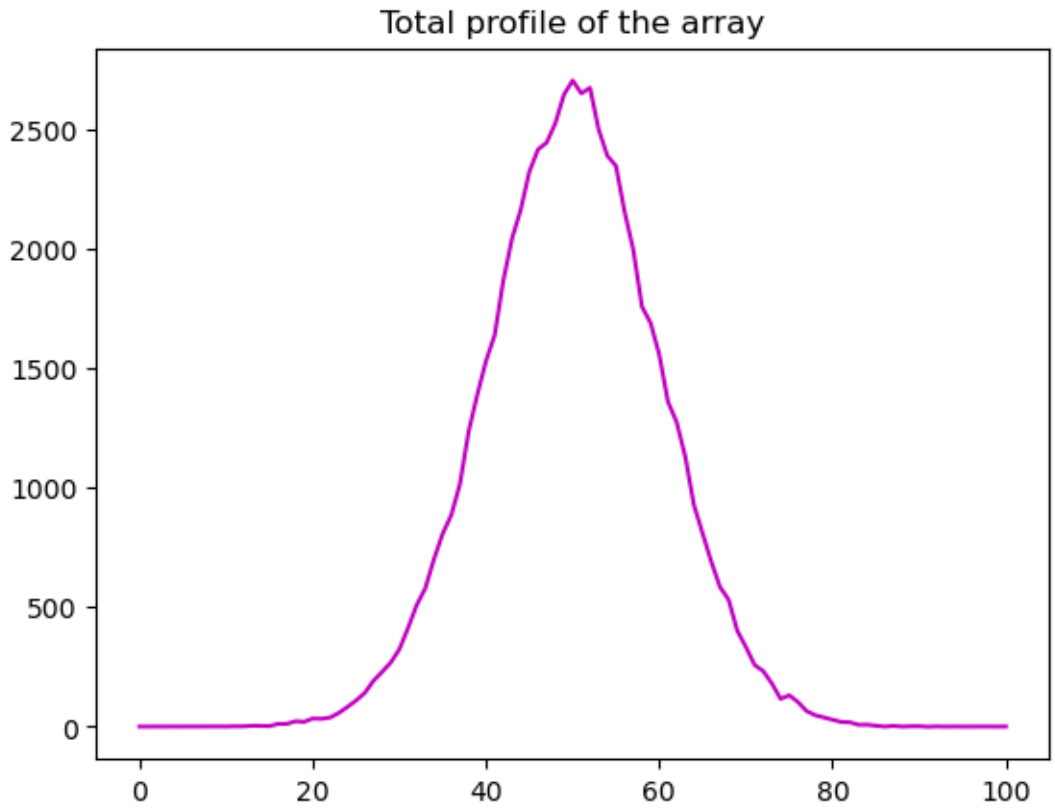


1.0.2 (b)

Generate a 1-D profile from your array. This can be either a single row of pixel values or a sum of rows. Then create a profile of the entire array, not just a single row.

[6]:
```
#generate and plot a 1D profile of the previous array
profile_single= zero_array[:, 50]
plt.plot(profile_single, color= 'm')
plt.title('1D-profile of the array')
plt.show()

#generate and plot the entire profile of the previous array
profile_total= zero_array.sum(axis=1)
plt.plot(profile_total, color= 'm')
plt.title('Total profile of the array')
plt.show()
```

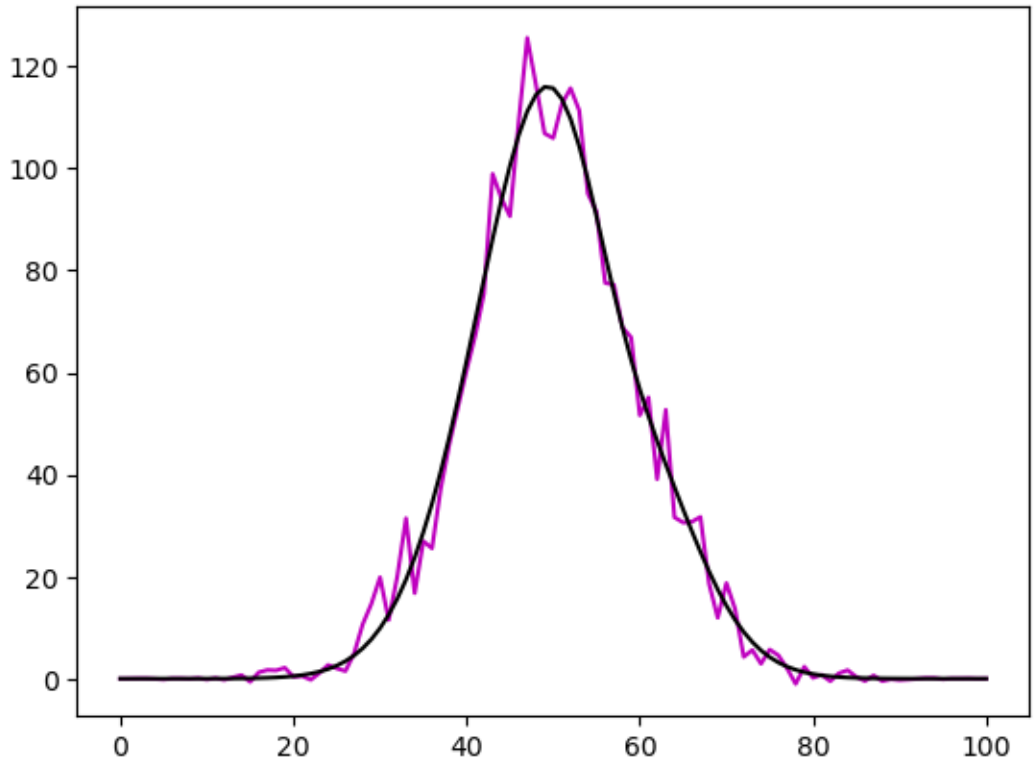

1D-profile of the array
120
100
80
60
40
20
0
0
20
40
60
80
100



1.0.3 (c)

Fit a double Gaussian to your 1-D profile, and plot the best model on top of the 1-D profile. This is hard. Skip this bit if needs be and come back to it.

[8]:
```
#define the double gaussian
def double_gauss_fit(x, center1, max_val1, sigma1, center2, max_val2, sigma2):
    first= max_val1*np.exp(-(x-center1)**2/(2*sigma1**2))
    second= max_val2*np.exp(-(x-center2)**2/(2*sigma2**2))
    return first+second

p0= (50, 90, 10, 60, 40, 3)
popt, pcov= curve_fit(double_gauss_fit, np.arange(profile_single.shape[0]),␣
 ↪profile_single, p0=p0)


#plot both the profile and the gaussian fit
plt.plot(profile_single, color= 'm')
plt.plot(double_gauss_fit(np.arange(profile_single.shape[0]),*popt), color= 'k')
plt.show()
```

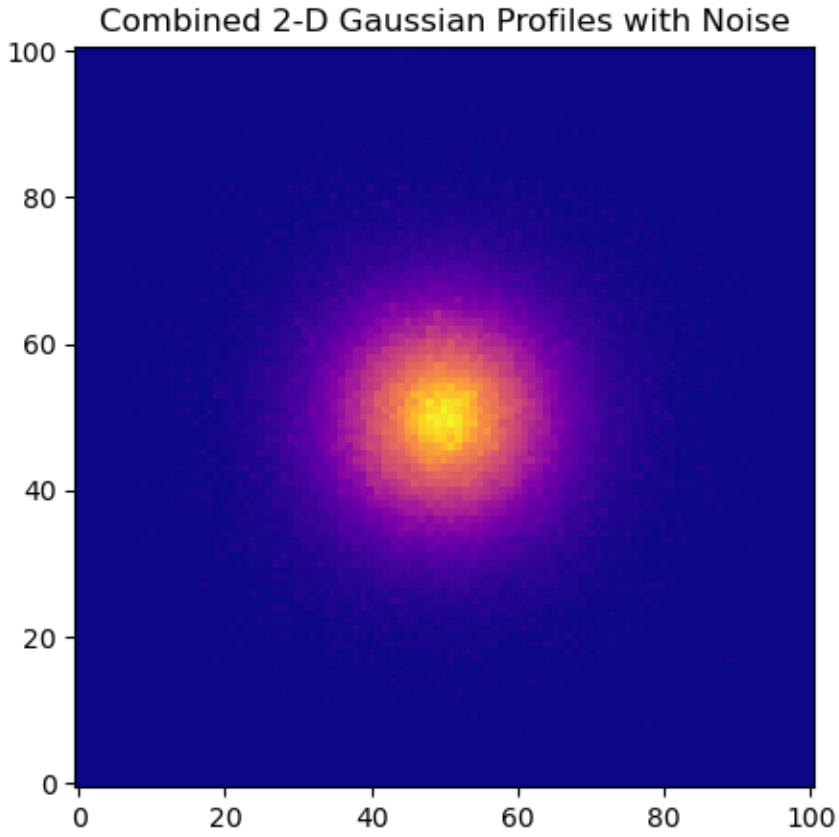


1.0.4 (d)

Repeat exercise (a), but use Poisson distribution for the background and the 2nd Gaussian (i.e. it is not possible for the value to drop below 0).

[10]:
```
#create the array with background noise
zero_array= np.zeros((101,101))
noise= np.random.normal(5, 3, zero_array.shape)

#add the 2D Gaussians with noise to the data
def gaussian(zero_array, max_value, center, sigma):
    x= np.linspace(0, zero_array.shape[0]-1, zero_array.shape[0])
    y= np.linspace(0, zero_array.shape[1]-1, zero_array.shape[1])
    x, y= np.meshgrid(x, y)
    gaussian_profile = max_value*np.exp(-((x-center[0])**2+(y-center[1])**2)/
 ↪(2*sigma**2))
    noise= np.random.poisson(np.sqrt(gaussian_profile))
    zero_array+= gaussian_profile+noise
    return zero_array

zero_array= gaussian(zero_array, max_value=100, center=(50, 50), sigma=10)
zero_array= gaussian(zero_array, max_value=20, center=(50, 50), sigma=3)
```

```
# Plot the result
plt.figure(figsize=(5, 5))
plt.imshow(zero_array, cmap='plasma', origin='lower')
plt.title('Combined 2-D Gaussian Profiles with Noise')
plt.show()
```



1.0.5 (e)

Consider a 101 by 101 array: + Generate a list of several thousand random locations within your such array, e.g. [(3,75),(56,34)] - Calculate how often one of your random locations falls within 10 pixels of any of the (straight) edges. * Calculate how often one of your random locations falls in either of these regions: a) outside a circle with radius 40 pixels, b) within 10 pixels of the inside of circle.

To check, probabilities should be approximately : a) 0.5, b) 0.22

[12]:
```
#create an array of random locations
num_loc= 2000
random_loc= [(random.randint(0, 100), random.randint(0, 100)) for i in␣
↪range(num_loc)]
```

```
#calculate how often the locations fall within 10 pixels
counter= 0
for x,y in random_loc:
    if (10<x<90) and (10<y<90):
        pass
    else:
        counter+= 1
print('How often one of the random locations falls within 10 pixels of any of␣
 ↪the straight edges:', counter)

#calculate how often the locations fall outside a circle of r= 40 pixels and the␣
 ↪probability
out_counter= 0
for x,y in random_loc:
    distance= np.sqrt((x-50)**2+(y-50)**2)
    if distance>40:
        pass
    else:
        out_counter+=1
print('How often one of the random locations falls outside a circle with radius␣
 ↪40 pixels:', out_counter)
print('Probability of how often one of the random locations falls outside a␣
 ↪circle with radius 40 pixels:', out_counter/num_loc)

#calculate how often the locations fall within 10 pixels of inside the circle␣
 ↪and the probability
in_counter= 0
for x,y in random_loc:
    distance= np.sqrt((x-50)**2+(y-50)**2)
    if 30<distance<=40:
        in_counter+=1
    else:
        pass
print('How often one of the random locations falls within 10 pixels of the␣
 ↪inside of the circle:', in_counter)
print('Probability of how often one of the random locations falls within 10␣
 ↪pixels of the inside of the circle:', in_counter/num_loc)
```

```
How often one of the random locations falls within 10 pixels of any of the
straight edges: 813
How often one of the random locations falls outside a circle with radius 40
pixels: 967
Probability of how often one of the random locations falls outside a circle with
radius 40 pixels: 0.4835
How often one of the random locations falls within 10 pixels of the inside of
the circle: 422
```

```
Probability of how often one of the random locations falls within 10 pixels of
the inside of the circle: 0.211
```

1.0.6 (f)

Ask the user to input a location and let them know if that location is in one of the two regions in (e)- ●●● . Allow repeat submissions, and allow the user to tell you when they want to move on.

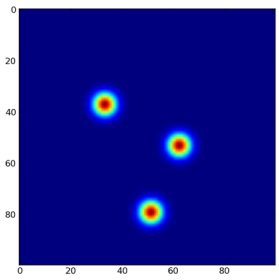
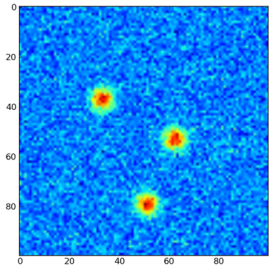
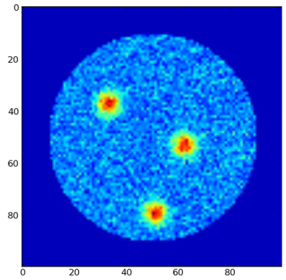
[14]:
```
#define a function to check the location
def check_loc(x, y):
    if (10<x<90) and (10<y<90):
        return 'The location falls within 10 pixels of any of the straight edges'

    distance= np.sqrt((x-50)**2+(y-50)**2)
    if distance>40:
        return 'The location falls outside a circle with radius 40 pixels'
    if 30<distance<=40:
        return 'The location falls within 10 pixels of the inside of the circle'

#define a function to let the user submit multiple entries
def main():
    while True:
        try:
            x= int(input('Enter a x integer value between 0 and 100: '))
            y= int(input('Enter a y integer value between 0 and 100: '))
            result= check_loc(x, y)
            print(result)

        except:
            print('Enter an integer value between 0 and 100')

        move_on= input('Do you want to input another location (yes or no)?: ').
 ↪strip().lower()
        if move_on != 'yes':
            break

if __name__== '__main__':
    main()
```

```
Enter a x integer value between 0 and 100:  67
Enter a y integer value between 0 and 100:  52

The location falls within 10 pixels of any of the straight edges

Do you want to input another location (yes or no)?:  no
```

1.0.7 (g)

Generate a 101 by 101 array, then: * add 3 synthetic "sources" by taking random locations and generating 2-D Gaussians around each one. Start with a peak value of 10 and a sigma of 3 (below

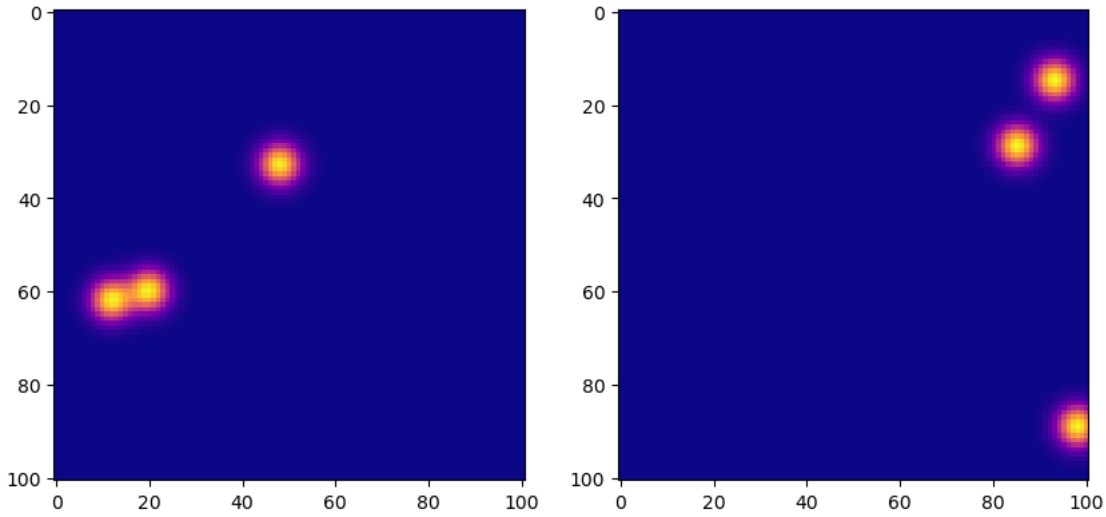
left). * Then add a noisy background that has a mean value of 3 and a sigma of 1 (below middle). * Then set all values (sources and background) to zero outside a circle of radius 40 pixels (below right). * Repeat 2 times (to make 6 images in total). Display all the images to the screen.

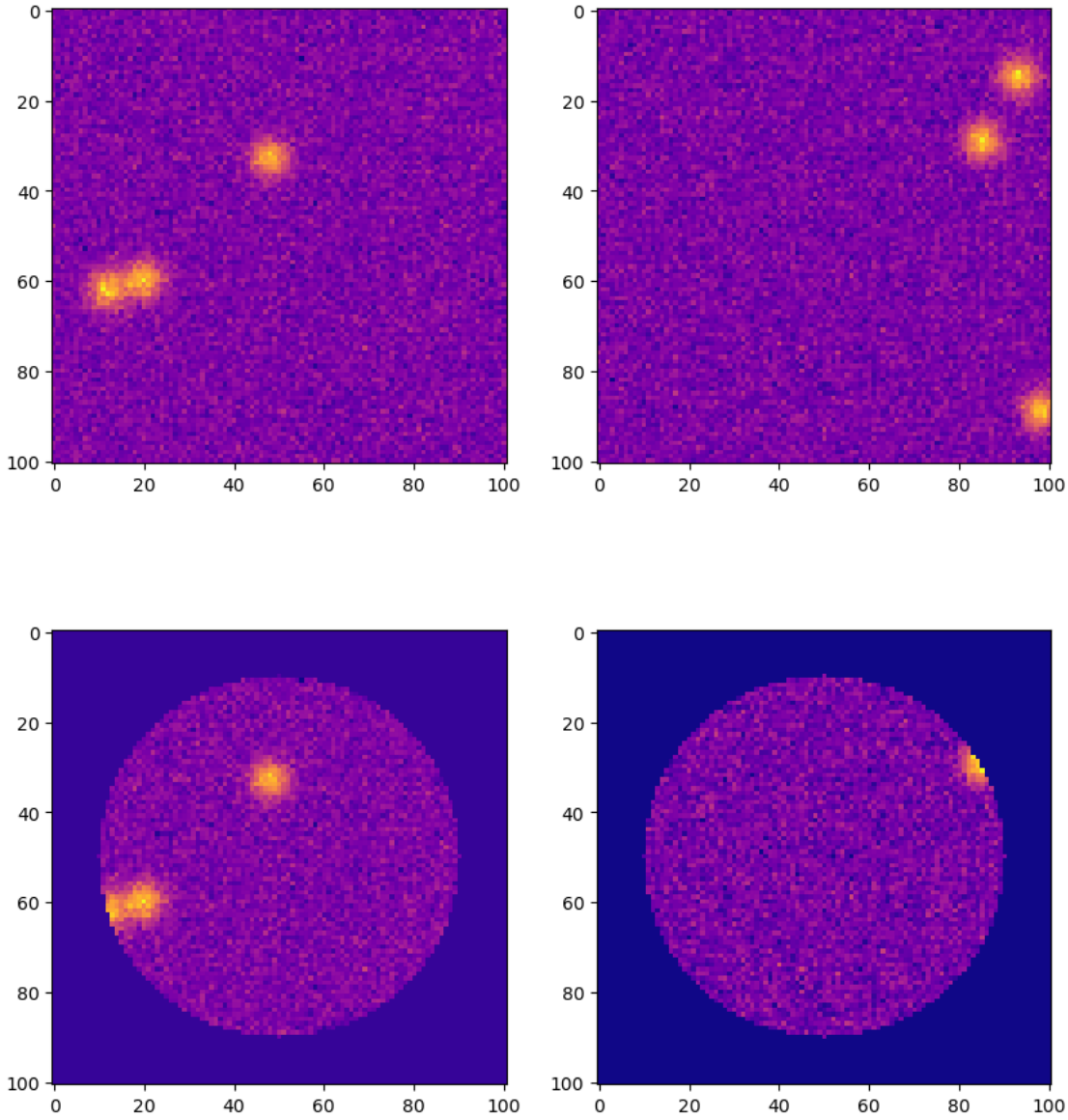




[16]:
```
#define the Gaussian function
def gaussian(zero_array, center_x, center_y, max_value, sigma):
    x= np.linspace(0, zero_array.shape[0]-1, zero_array.shape[0])
    y= np.linspace(0, zero_array.shape[1]-1, zero_array.shape[1])
    x, y= np.meshgrid(x, y)
    gaussian_profile= max_value*np.exp(-((x-center_x)**2+(y-center_y)**2)/
 ↪(2*sigma**2))
    zero_array+= gaussian_profile
    return zero_array


max_value= 10
sigma= 3
radius= 40

#define a function to create and store the source images
def source_images():
    zero_array= np.zeros((101, 101))
    source_pos= []
    for i in range(3):
        center_x= np.random.randint(0, 101)
        center_y= np.random.randint(0, 101)
        source_pos.append((center_x, center_y))
        gaussian(zero_array, center_x, center_y, max_value, sigma)
    return zero_array, source_pos

#define a function to add background noise
def noise(zero_array):
    background_noise= np.random.normal(3, 1, zero_array.shape)
    return zero_array+background_noise

#define a function to apply a circular mask
def circle(zero_array):
    y, x= np.ogrid[:101, :101]
```

```
    center= (101//2, 101//2)
    mask= (x-center[0])**2+(y-center[1])**2<= radius**2
    zero_array[~mask]= 0
    return zero_array

#define a function to check the location of the sources
def check_source_pos(source_pos):
    center= (101//2, 101//2)
    for (cx, cy) in source_pos:
        dist_to_center= np.sqrt((cx-center[0])**2+(cy-center[1])**2)
        if radius-10 <=dist_to_center<= radius:
            print(f'Source at ({cx}, {cy}) is within 10 pixels of the inside␣
 ↪edge of the circle')
        elif dist_to_center> radius:
            print(f'Source at ({cx}, {cy}) is outside the circle in the zero␣
 ↪region')
        elif 0 <=dist_to_center< radius-10:
            print(f'Source at ({cx}, {cy}) is in none of the required locations')


#define a function to check the percentage of the source falling in the zero␣
 ↪region
def signal_outside_circle(source_pos2, center, radius, zero_array, sigma):
    for (cx, cy) in source_pos2:
        total_signal= 2*np.pi*sigma**2
        y, x= np.ogrid[:zero_array.shape[0], :zero_array.shape[1]]
        dist_to_center= np.sqrt((x-center[0])**2 + (y-center[1])**2)
        mask_outside= dist_to_center> radius
        gaussian_profile= np.exp(-((x-cx)**2 + (y-cy)**2)/(2*sigma**2))
        signal_outside= np.sum(gaussian_profile[mask_outside])
        percentage_outside= (signal_outside/total_signal)*100
        print(f'Source at ({cx}, {cy}) has {percentage_outside:.2f}% of its␣
 ↪signal outside the circle')


images_with_sources= []
source_pos_list= []
for i in range(2):
    img, source_pos= source_images()
    images_with_sources.append(img)
    source_pos_list.append(source_pos)

#plot the images with just the sources
plt.figure(figsize= (10, 5))
for i, img in enumerate(images_with_sources):
    plt.subplot(1, 2, i+1)
    plt.imshow(img, cmap= 'plasma')
```

```
plt.show()


#plot the images with the background
images_with_background= [noise(img.copy()) for img in images_with_sources]
plt.figure(figsize= (10, 5))
for i, img in enumerate(images_with_background):
    plt.subplot(1, 2, i+1)
    plt.imshow(img, cmap= 'plasma')
plt.show()


#plot the images with the circular mask
final_images= [circle(img.copy()) for img in images_with_background]
plt.figure(figsize= (10, 5))
for i, img in enumerate(final_images):
    plt.subplot(1, 2, i+1)
    plt.imshow(img, cmap='plasma')
plt.show()
```





1.0.8 (h)

Run through each of your six images and alert the user if any of the sources therein falls within 10 pixels of the inside edge of your 40 pixel circle or in the "zero'd" region outside. [You will have to preserve the locations you used to generate the 6x3 sources]

[18]:
```
for i, source_pos in enumerate(source_pos_list):
    print(f'For image {i+1}:')
    check_source_pos(source_pos)
```

```
For image 1:
Source at (20, 60) is within 10 pixels of the inside edge of the circle
```

```
Source at (48, 33) is in none of the required locations
Source at (12, 62) is within 10 pixels of the inside edge of the circle
For image 2:
Source at (98, 89) is outside the circle in the zero region
Source at (93, 15) is outside the circle in the zero region
Source at (85, 29) is outside the circle in the zero region
```

1.0.9 (i)

Tell the user what percentage of the signal from each source fell into the "zero" region outside the circle. For that last bit, you can use your knowledge of the source location, normalisation and sigma [look up "volume under the gaussian function" in Wikipedia].

[20]:
```
center= (101//2, 101//2)
radius= 40
sigma= 3

zero_array, _= source_images()
source_pos2= sum(source_pos_list, [])

signal_outside_circle(source_pos2, center, radius, zero_array, sigma)
```

```
Source at (20, 60) has 0.29% of its signal outside the circle
Source at (48, 33) has 0.00% of its signal outside the circle
Source at (12, 62) has 49.52% of its signal outside the circle
Source at (98, 89) has 79.87% of its signal outside the circle
Source at (93, 15) has 99.40% of its signal outside the circle
Source at (85, 29) has 60.88% of its signal outside the circle
```

1.0.10 (j)

Switch to circularly symmetric "Beta-models' ' (see equation below) for your shapes and repeat steps (g),(h),(i)

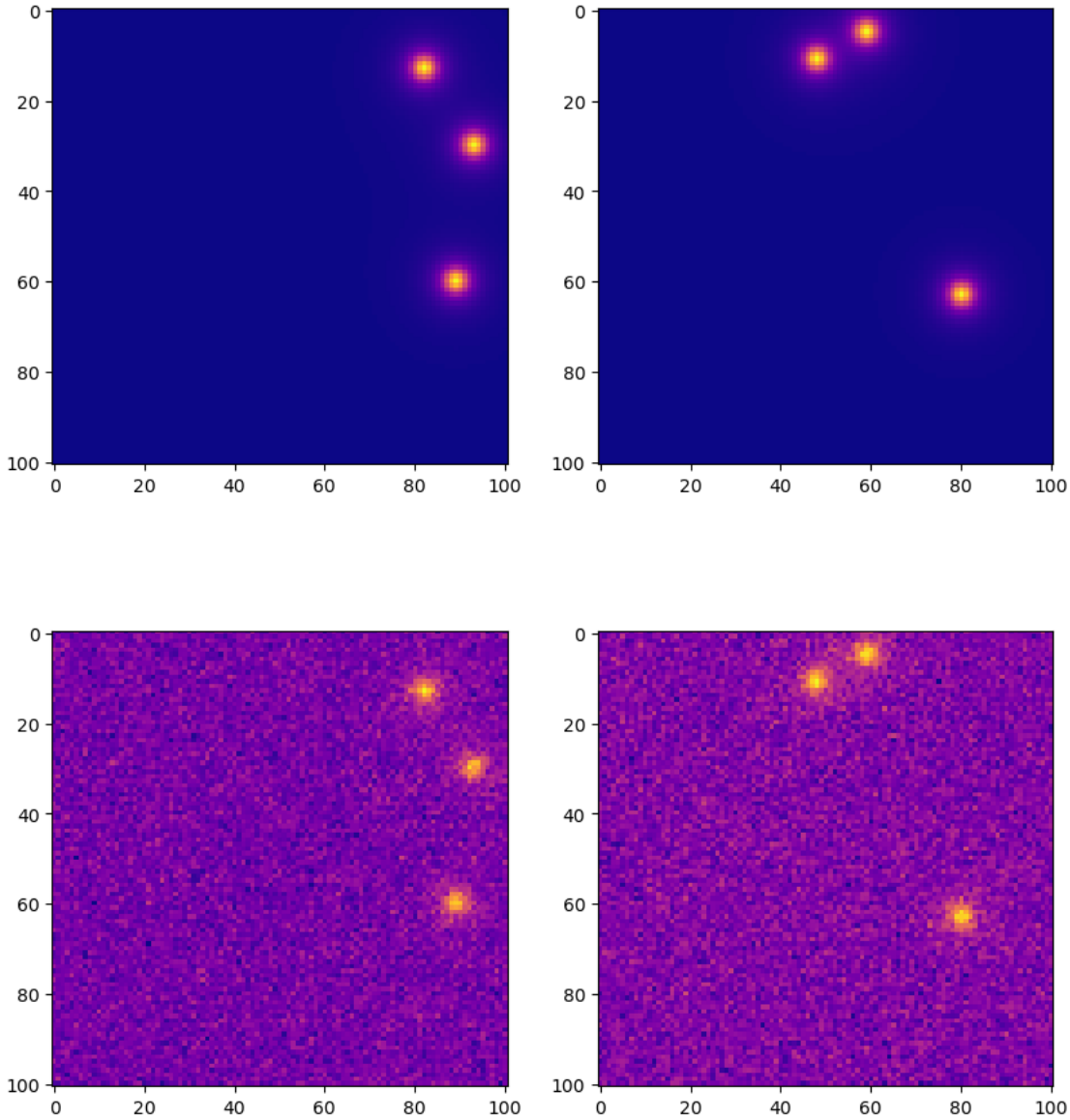
$$y = S_0(1 + (r/r_0)^2)^{-1.5}$$

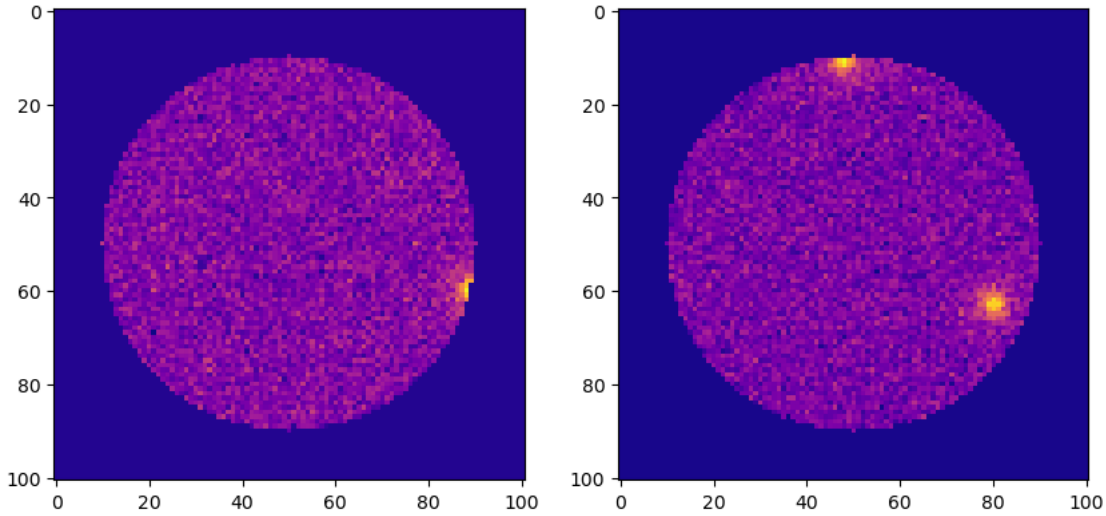
where r is the radius from the centre, r_0 is the "core" [use r_0=3 for now], S_0 is the normalisation [use 10 for now]. Note that the equivalent of step (i) is harder than before because you will need to numerically integrate [rather than using an analytical formula]

[22]:
```
#create an array with 3 random sources and a beta model
def beta_model(zero_array, center_x, center_y, S0, r0):
    x= np.linspace(0, zero_array.shape[0]-1, zero_array.shape[0])
    y= np.linspace(0, zero_array.shape[1]-1, zero_array.shape[1])
    x, y= np.meshgrid(x, y)
    r= np.sqrt((x-center_x)**2+(y-center_y)**2)
    beta= S0*(1+(r/r0)**2)**-1.5
    zero_array+= beta
```

```
S0= 10
r0= 3
radius= 40
#define a function to create and store the source images
def source_images():
    zero_array= np.zeros((101, 101))
    source_pos= []
    for i in range(3):
        center_x= np.random.randint(0, 101)
        center_y= np.random.randint(0, 101)
        source_pos.append((center_x, center_y))
        beta_model(zero_array, center_x, center_y, S0, r0)
    return zero_array, source_pos

#define a function to add background noise
def noise(zero_array):
    background_noise= np.random.normal(3, 1, zero_array.shape)
    return zero_array+background_noise

#define a function to apply a circular mask
def circle(zero_array):
    y, x= np.ogrid[:101, :101]
    center= (101//2, 101//2)
    mask= (x-center[0])**2+(y-center[1])**2<= radius**2
    zero_array[~mask]= 0
    return zero_array

#define a function to check the location of the sources
def check_source_pos(source_pos):
    center= (101//2, 101//2)
    for (cx, cy) in source_pos:
        dist_to_center= np.sqrt((cx-center[0])**2+(cy-center[1])**2)
        if radius-10 <=dist_to_center<= radius:
            print(f'Source at ({cx}, {cy}) is within 10 pixels of the inside␣
 ↪edge of the circle')
        elif dist_to_center> radius:
            print(f'Source at ({cx}, {cy}) is outside the circle in the zero␣
 ↪region')
        elif 0 <=dist_to_center< radius-10:
            print(f'Source at ({cx}, {cy}) is in none of the required locations')

#define a function to check the percentage of the source falling in the zero␣
 ↪region
def signal_outside_circle(source_pos2, center, radius, zero_array, r0):
    for (cx, cy) in source_pos2:
        total_signal= 2*np.pi*sigma**2
```

```
        y, x= np.ogrid[:zero_array.shape[0], :zero_array.shape[1]]
        dist_to_center= np.sqrt((x-center[0])**2 + (y-center[1])**2)
        mask_outside= dist_to_center> radius
        gaussian_profile= np.exp(-((x-cx)**2 + (y-cy)**2)/(2*sigma**2))
        signal_outside= np.sum(gaussian_profile[mask_outside])
        percentage_outside= (signal_outside/total_signal)*100
        print(f'Source at ({cx}, {cy}) has {percentage_outside:.2f}% of its␣
 ↪signal outside the circle')


images_with_sources= []
source_pos_list= []
for i in range(2):
    img, source_pos= source_images()
    images_with_sources.append(img)
    source_pos_list.append(source_pos)


#plot the images with just the sources
plt.figure(figsize= (10, 5))
for i, img in enumerate(images_with_sources):
    plt.subplot(1, 2, i+1)
    plt.imshow(img, cmap= 'plasma')
plt.show()


#plot the images with the background
images_with_background= [noise(img.copy()) for img in images_with_sources]
plt.figure(figsize= (10, 5))
for i, img in enumerate(images_with_background):
    plt.subplot(1, 2, i+1)
    plt.imshow(img, cmap= 'plasma')
plt.show()


#plot the images with the circular mask
final_images= [circle(img.copy()) for img in images_with_background]
plt.figure(figsize= (10, 5))
for i, img in enumerate(final_images):
    plt.subplot(1, 2, i+1)
    plt.imshow(img, cmap='plasma')
plt.show()
```


0
20
40
60
80
100
0
20
40
60
80
100



[23]:
```
for i, source_pos in enumerate(source_pos_list):
    print(f'For image {i+1}:')
    check_source_pos(source_pos)
```

```
For image 1:
Source at (93, 30) is outside the circle in the zero region
Source at (82, 13) is outside the circle in the zero region
Source at (89, 60) is outside the circle in the zero region
For image 2:
Source at (59, 5) is outside the circle in the zero region
Source at (48, 11) is within 10 pixels of the inside edge of the circle
Source at (80, 63) is within 10 pixels of the inside edge of the circle
```

[24]:
```
center= (101//2, 101//2)
radius= 40
r0= 3

zero_array, _= source_images()
source_pos2= sum(source_pos_list, [])
signal_outside_circle(source_pos2, center, radius, zero_array, sigma)
```

```
Source at (93, 30) has 98.76% of its signal outside the circle
Source at (82, 13) has 99.86% of its signal outside the circle
Source at (89, 60) has 55.48% of its signal outside the circle
Source at (59, 5) has 94.58% of its signal outside the circle
Source at (48, 11) has 42.21% of its signal outside the circle
Source at (80, 63) has 0.79% of its signal outside the circle
```

1.0.11 (k)

Produce an image "with vignetting": i.e. now the sensitivity of the "detector" is decreasing radially from 100% in the centre to 50% at the far corner. The functional form of this decrease is linear. Note that your background is not vignetted (so the expectation value does not change across the detector).

[ ]:

1.0.12 (l)

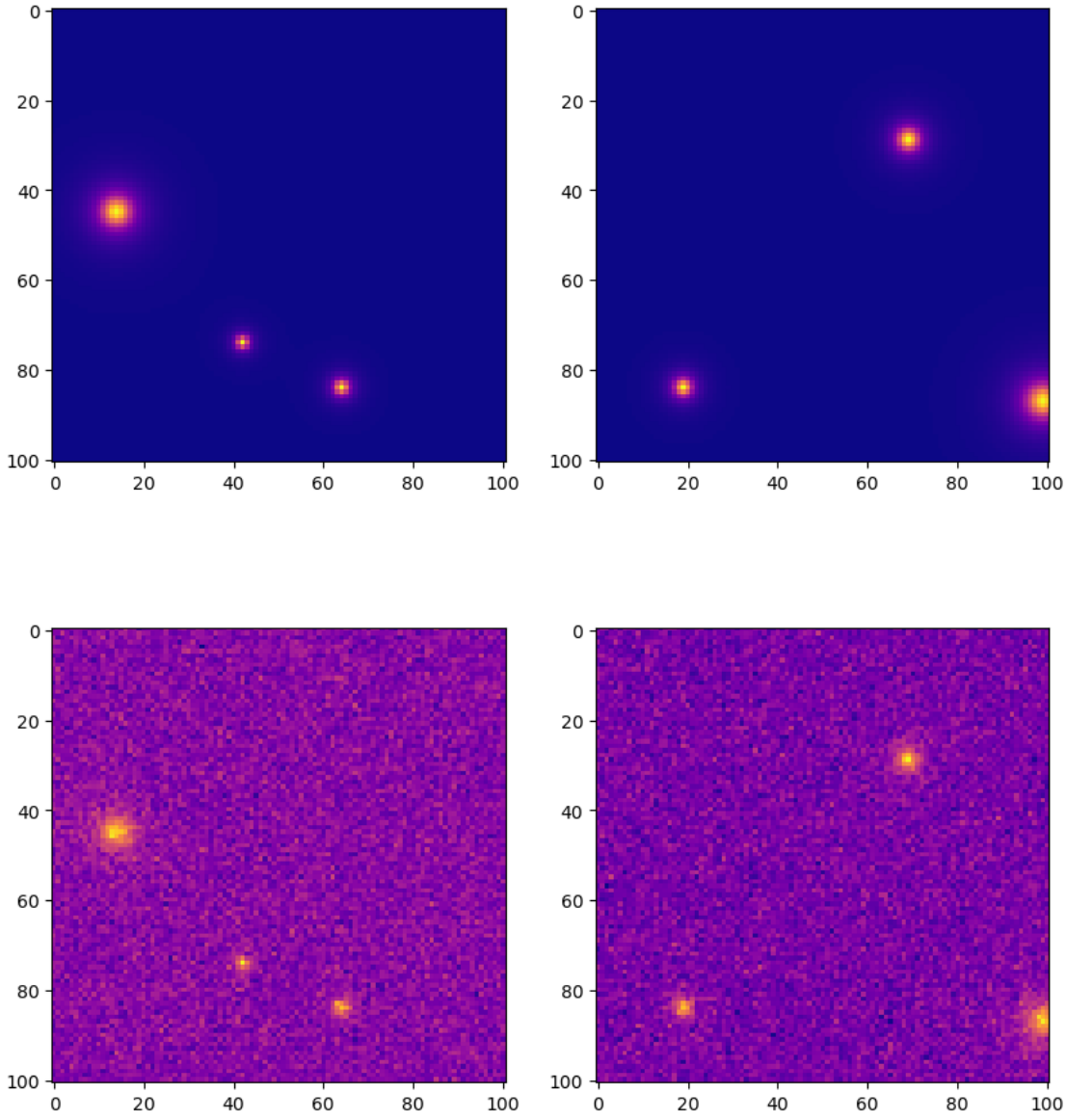
repeat (j) when $1 < r_0 < 5$ with a flat (constant) probability distribution and a well-known center. Since we don't know what our value for r_0 is, we will make the assumption that r_0=3. Return the error in making this assumption.

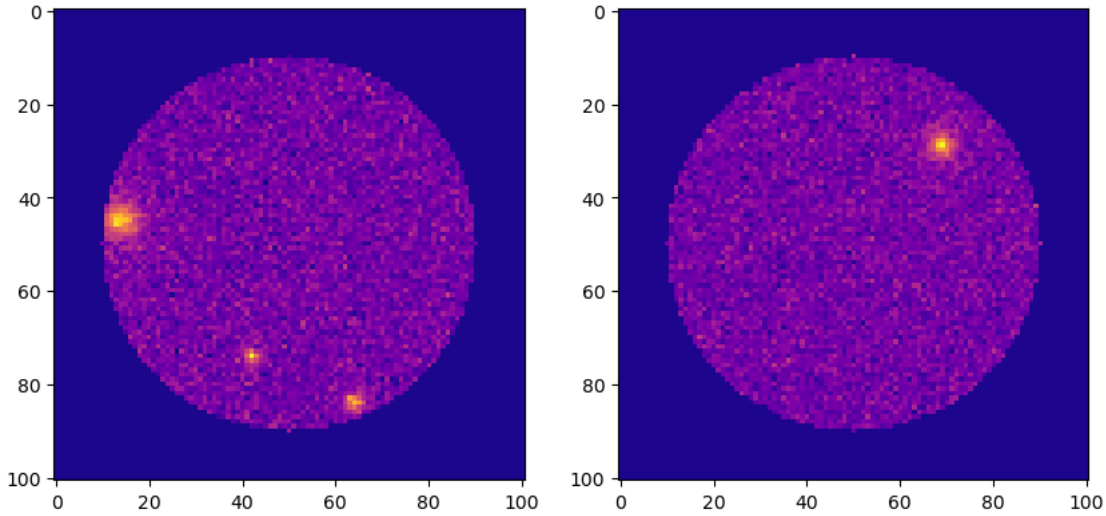
[27]:
```
#create an array with 3 random sources and a beta model
def beta_model(zero_array, center_x, center_y, S0, r0):
    x= np.linspace(0, zero_array.shape[0]-1, zero_array.shape[0])
    y= np.linspace(0, zero_array.shape[1]-1, zero_array.shape[1])
    x, y= np.meshgrid(x, y)
    r= np.sqrt((x-center_x)**2+(y-center_y)**2)
    beta= S0*(1+(r/r0)**2)**-1.5
    zero_array+= beta


S0= 10
assumed_r0= 3
radius= 40
#define a function to create and store the source images
def source_images():
    zero_array= np.zeros((101, 101))
    source_pos= []
    r0_values= []
    for i in range(3):
        center_x= np.random.randint(0, 101)
        center_y= np.random.randint(0, 101)
        r0= np.random.uniform(1, 5)
        r0_values.append(r0)
        source_pos.append((center_x, center_y))
        beta_model(zero_array, center_x, center_y, S0, r0)
    return zero_array, source_pos, r0_values

#define a function to add background noise
def noise(zero_array):
    background_noise= np.random.normal(3, 1, zero_array.shape)
    return zero_array+background_noise

#define a function to apply a circular mask
```

```
def circle(zero_array):
    y, x= np.ogrid[:101, :101]
    center= (101//2, 101//2)
    mask= (x-center[0])**2+(y-center[1])**2<= radius**2
    zero_array[~mask]= 0
    return zero_array

#define a function to calculate the error of the r0 value
def calculate_error(r0_values, assumed_r0):
    errors= []
    for actual_r0 in r0_values:
        error = round(abs((actual_r0-assumed_r0)/actual_r0)*100, 2)  #␣
 ↪Percentage error
        errors.append(error)
    return errors


#define a function to check the location of the sources
def check_source_pos(source_pos):
    center= (101//2, 101//2)
    for (cx, cy) in source_pos:
        dist_to_center= np.sqrt((cx-center[0])**2+(cy-center[1])**2)
        if radius-10 <=dist_to_center<= radius:
            print(f'Source at ({cx}, {cy}) is within 10 pixels of the inside␣
 ↪edge of the circle')
        elif dist_to_center> radius:
            print(f'Source at ({cx}, {cy}) is outside the circle in the zero␣
 ↪region')
        elif 0 <=dist_to_center< radius-10:
            print(f'Source at ({cx}, {cy}) is in none of the required locations')

#define a function to check the percentage of the source falling in the zero␣
 ↪region
def signal_outside_circle(source_pos2, center, radius, zero_array, r0):
    for (cx, cy) in source_pos2:
        total_signal= 2*np.pi*sigma**2
        y, x= np.ogrid[:zero_array.shape[0], :zero_array.shape[1]]
        dist_to_center= np.sqrt((x-center[0])**2 + (y-center[1])**2)
        mask_outside= dist_to_center> radius
        gaussian_profile= np.exp(-((x-cx)**2 + (y-cy)**2)/(2*sigma**2))
        signal_outside= np.sum(gaussian_profile[mask_outside])
        percentage_outside= (signal_outside/total_signal)*100
        print(f'Source at ({cx}, {cy}) has {percentage_outside:.2f}% of its␣
 ↪signal outside the circle')


images_with_sources= []
```

```
source_pos_list= []
r0_values_list= []
for i in range(2):
    img, source_pos, r0_values= source_images()
    images_with_sources.append(img)
    source_pos_list.append(source_pos)
    r0_values_list.append(r0_values)


#plot the images with just the sources
plt.figure(figsize= (10, 5))
for i, img in enumerate(images_with_sources):
    plt.subplot(1, 2, i+1)
    plt.imshow(img, cmap= 'plasma')
plt.show()


#plot the images with the background
images_with_background= [noise(img.copy()) for img in images_with_sources]
plt.figure(figsize= (10, 5))
for i, img in enumerate(images_with_background):
    plt.subplot(1, 2, i+1)
    plt.imshow(img, cmap= 'plasma')
plt.show()


#plot the images with the circular mask
final_images= [circle(img.copy()) for img in images_with_background]
plt.figure(figsize= (10, 5))
for i, img in enumerate(final_images):
    plt.subplot(1, 2, i+1)
    plt.imshow(img, cmap='plasma')
plt.show()


# Check source positions
for i, source_pos in enumerate(source_pos_list):
    print(f'For image {i+1}:')
    check_source_pos(source_pos)


center= (101//2, 101//2)
radius=40
for i, source_pos in enumerate(source_pos_list):
    print(f'For image {i+1}:')
    signal_outside_circle(source_pos, center, radius, final_images[i], sigma)
```

```
# Calculate error for each image
for i, r0_values in enumerate(r0_values_list):
    errors= calculate_error(r0_values, assumed_r0)
    print(f'The error percentage for each source in image {i+1} are: {errors}')
```






```
For image 1:
Source at (64, 84) is within 10 pixels of the inside edge of the circle
Source at (42, 74) is in none of the required locations
Source at (14, 45) is within 10 pixels of the inside edge of the circle
For image 2:
Source at (19, 84) is outside the circle in the zero region
Source at (69, 29) is in none of the required locations
Source at (99, 87) is outside the circle in the zero region
For image 1:
Source at (64, 84) has 14.56% of its signal outside the circle
Source at (42, 74) has 0.00% of its signal outside the circle
Source at (14, 45) has 12.92% of its signal outside the circle
For image 2:
Source at (19, 84) has 97.85% of its signal outside the circle
Source at (69, 29) has 0.01% of its signal outside the circle
Source at (99, 87) has 69.23% of its signal outside the circle
The error percentage for each source in image 1 are: [75.57, 114.84, 17.35]
The error percentage for each source in image 2 are: [49.18, 16.03, 24.94]
```

1.0.13 (m)

similar to (l), but now you can know what your r0 value is, but you don't know where the centre is. You can guess the centre using the brightest pixel in the "active" area, i.e. inside our circle. Return the error on your percentage of lost flux.