CSCI E-25

Filtering Images

Steve Elston

Introduction

Preprocessing of images is generally an essential step in most computer vision applications. Preprocessing typically involves filtering of one type or another which is out focus here. Here we will focus on two types of filters.

1. Filters for noise elimination and edge detection.
2. Morphology filters.

To get started execute the code in the cell below to import the packages you will need.

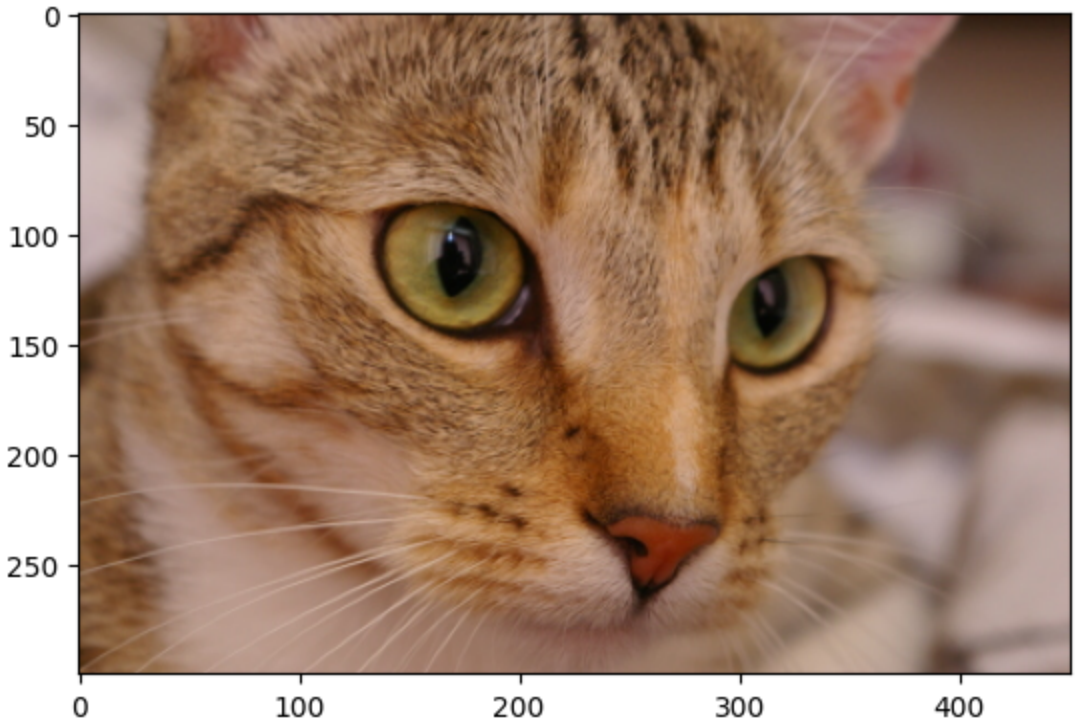
In [ ]:
```
# Libraries
import skimage
from skimage import data
from skimage.filters.rank import equalize
import skimage.filters as skfilters
import skimage.morphology as morphology
import skimage.transform as transform
from skimage.color import rgb2gray
from skimage import exposure
from skimage.transform import pyramid_gaussian
import numpy as np
from scipy import fft
from matplotlib.colors import LogNorm
import matplotlib.pyplot as plt
%matplotlib inline
```

Load and Prepare Image

For the exercises you will work with a complex image of a striped cat. Execute the code in the cell below to load and display the image.

In [ ]:
```
cat_image = data.cat()
print('Image size = ' + str(cat_image.shape))
_ = plt.imshow(cat_image)
```

```
Image size = (300, 451, 3)
```

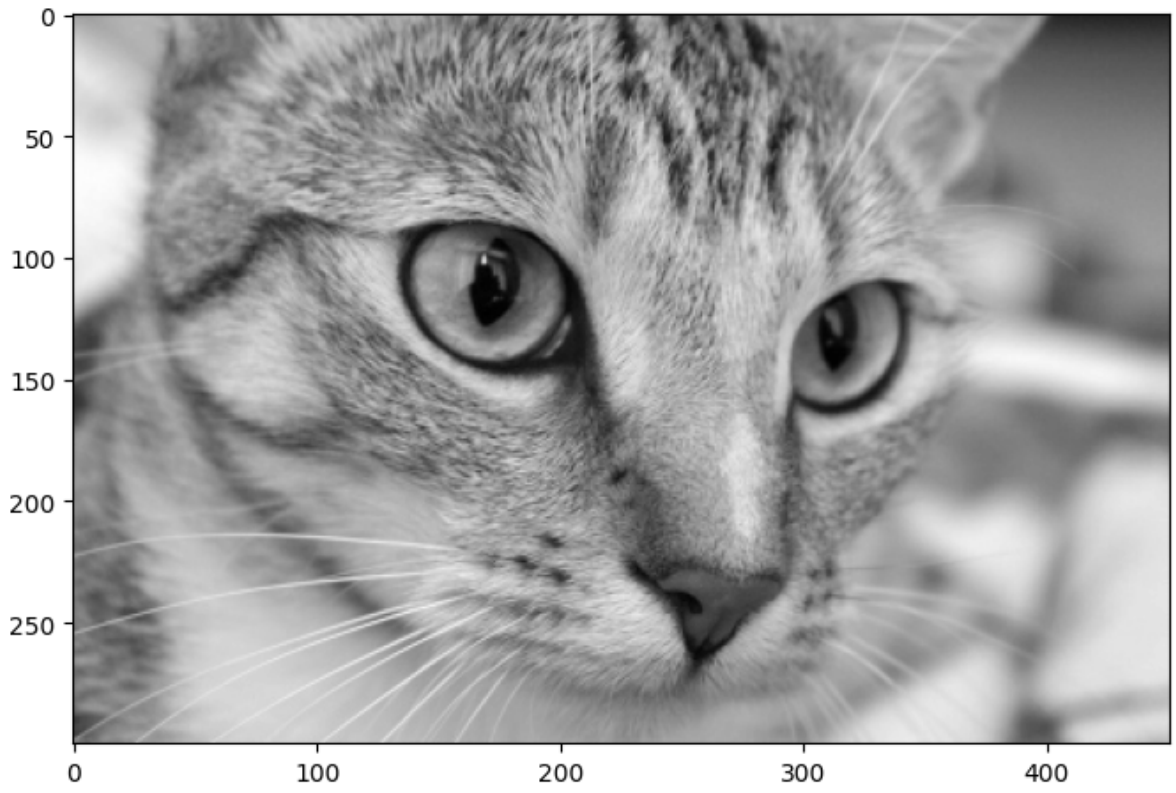


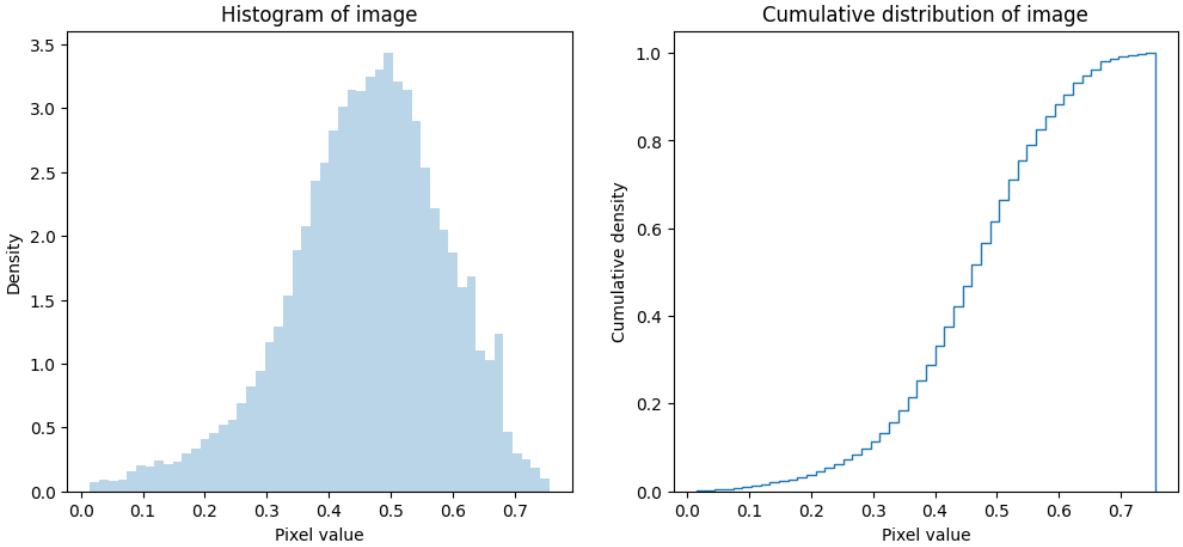
This image is complex. The image shows typical typical facial features of a cat, eyes, nose, whiskers, etc. The cat's face also has a complex pattern of strips and patches of different colors. Additionally, there are complex features in the background, including the cat's chest patch, part of an ear and blurred patches that appear to include the cat's flank. Preparing such a complex image for further processing can be quite challenging.

> **Exercise 2-1:** You will now prepare the cat image for further processing by the following steps:
>
> 1. Convert the 3-channel color image to gray-scale.
> 2. Display the gray-scale image along with the distribution of pixel values using the functions provided.
> 3. Apply adaptive histogram equalization to the gray-scale image. Name the resulting image `cat_grayscale_equalized`.
> 4. Display the equalized gray-scale image along with the distribution of pixel values using the functions provided.

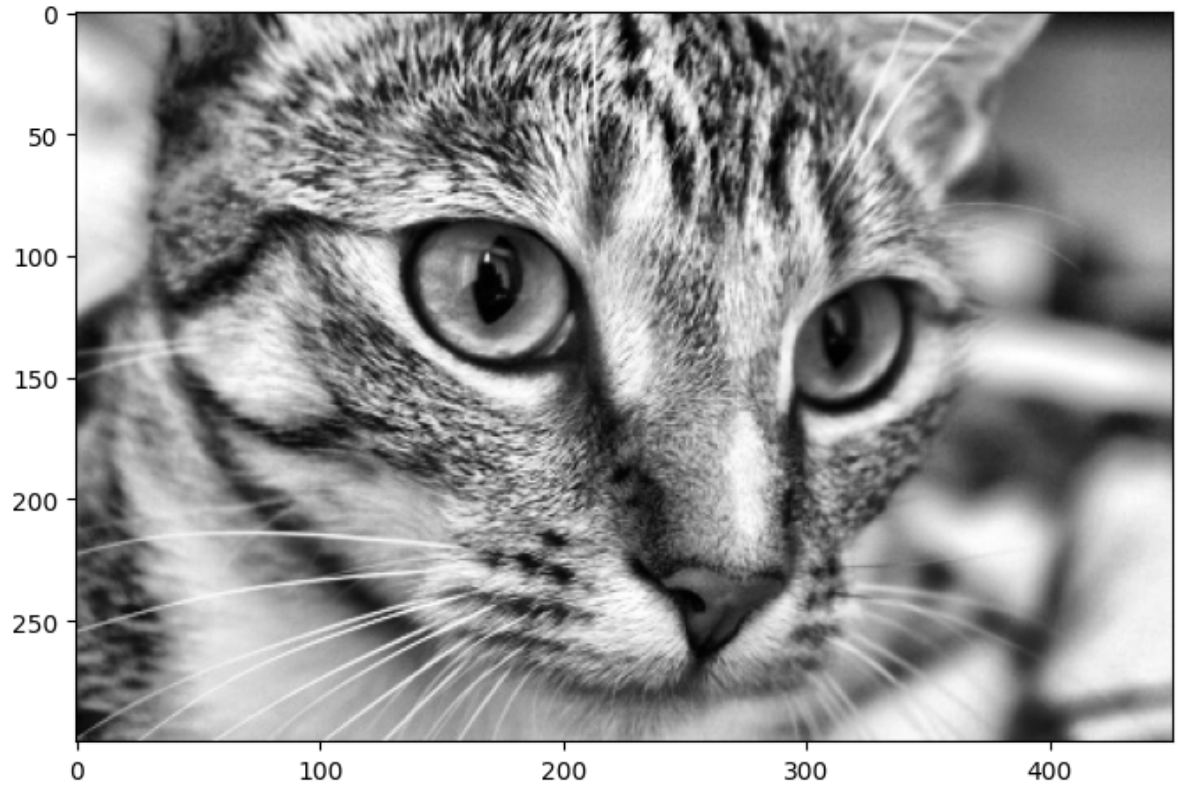
In [ ]:
```
def plot_grayscale(img, h = 8):
    plt.figure(figsize=(h, h))
    _ = plt.imshow(img, cmap = plt.get_cmap('gray'))
    plt.show()

def plot_gray_scale_distribution(img):
    '''Function plots histograms a gray scale image along
    with the cumulative distribution'''
    fig, ax = plt.subplots(1, 2, figsize=(12, 5))
    ax[0].hist(img.flatten(), bins = 50, density = True, alpha = 0.3)
```

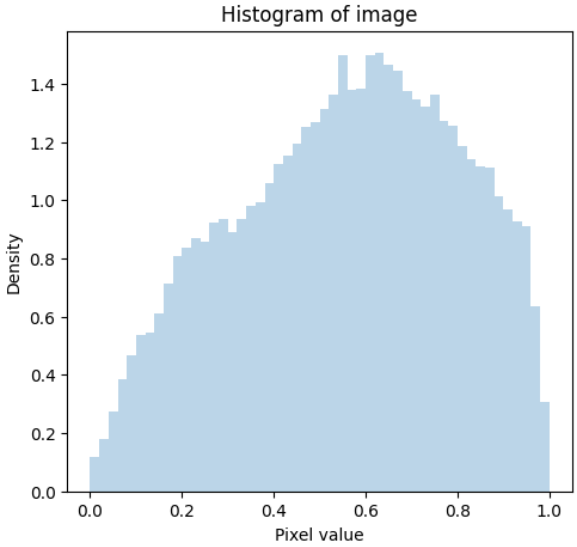
```
    ax[0].set_title('Histogram of image')
    ax[0].set_xlabel('Pixel value')
    ax[0].set_ylabel('Density')
    ax[1].hist(img.flatten(), bins = 50, density = True, cumulative = True,
    ax[1].set_title('Cumulative distribution of image')
    ax[1].set_xlabel('Pixel value')
    ax[1].set_ylabel('Cumulative density')
    plt.show()

## Put your code below

# 1. Convert the 3-channel color image to gray-scale.
cat_grayscale = rgb2gray(cat_image)

# 2. Display the gray-scale image along with the distribution of pixel value
plot_grayscale(cat_grayscale)
plot_gray_scale_distribution(cat_grayscale)
```

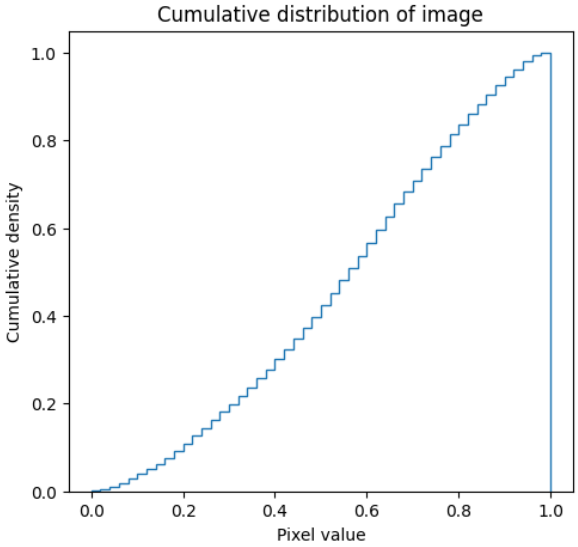




In [ ]:
```
## Put your code below

def equalize_image(img, func):
    '''equalize_image equalizes the image img using the function passed as a
    The function must be a function from skimage.filters.rank.
    The function returns the equalized image.

    To be used with any of these functions:
    - skimage.exposure.equalize_hist
    - skimage.exposure.equalize_adapthist
    '''

    img_equalized = eval(func)(img)
    return img_equalized


def plot_grayscale_equalized(img_equalized):
    plot_grayscale(img_equalized)
    plot_gray_scale_distribution(img_equalized)

# 3. Apply adaptive histogram equalization to the gray-scale image. Name the
#    `cat_grayscale_equalized`.
cat_grayscale_equalized = equalize_image(cat_grayscale, 'skimage.exposure.ec

# 4. Display the equalized gray-scale image along with the distribution of p
#    the functions provided.
plot_grayscale_equalized(cat_grayscale_equalized)
```







In [ ]:
```
# Equalization test
print(cat_grayscale_equalized.dtype)
print(np.min(cat_grayscale_equalized))
print(np.max(cat_grayscale_equalized))
```

```
float64
0.0
1.0
```

> Answer the following questions:
>
> 1. How has the adaptive histogram equalization changed the image and is this change an improvement from a computer vision point of view?

> 2. How has the adaptive histogram equalization changed the distribution of pixel values?
>
> **End of exercise.**

> **Answers:**

> 1. Adaptive histogram equalization has improved the image by increasing the contrast of the image. This is an improvement from a computer vision point of view because it will make it easier to detect edges and other features. A clear example is the cat's nose and mouth. The nose and mouth are much easier to see in the equalized image.

> 2. The distribution of pixel values in adaptive histogram equalization shows more pixels with higher values, while the density is also adjusted to lower values. The cumulative distribution became a straight line. This is because the histogram equalization is applied to small regions of the image, and the contrast is enhanced in each region.

> **Important note!:** For subsequent exercises, unless otherwise stated, your starting point should be the `cat_grayscale_equalized` image just created. This is in keeping with good practice of starting with a properly adjusted image before applying any CV algorithms.

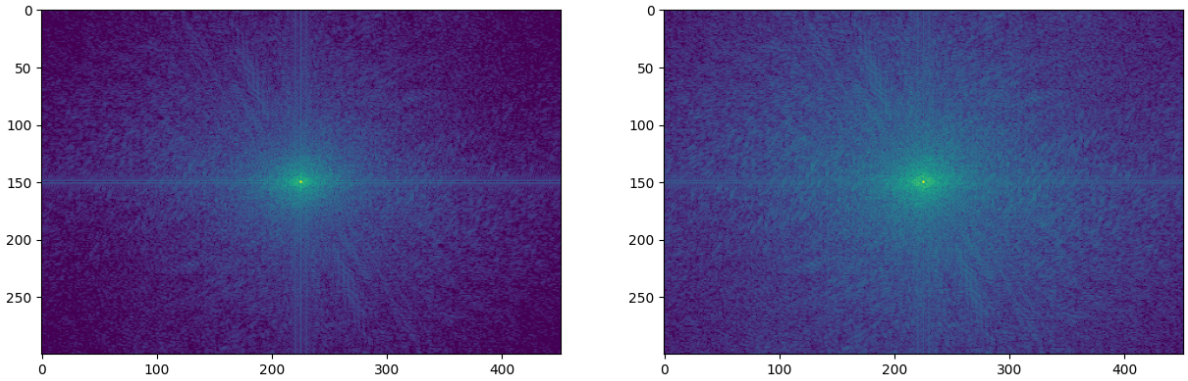
Another way to understand the effect of a filter on the image is to examine the power spectrum of the image. Specifically, we will use the scipy.fft.fft2 function which performs a d-dimensional fast Fourier transform (FFT) on the real-valued image. The zero frequency components of the FFT are shifted to the center of the array using the scipy.fft.fftshift function.

The function in the cell below computes the FFT of two images. The results of the FFT calculation are complex numbers, amplitude and phase. A 2-dimensional array of complex numbers is hard to display, so we will use the magnitude of the complex number, using the numpy.absolute function. The values are then squared to units of power using the numpy.square function.

These values can range over several orders of magnitude so we use a logarithmic color scale for the image display.

To see the difference in the frequency components of the original image and the equalized image, execute the code in the cell below.

In [ ]:
```
image_power = lambda x: np.square(np.abs(fft.fftshift(fft.fft2(x))))

def plot_2d_fft(img1, img2):
    fig, ax = plt.subplots(1, 2, figsize=(15, 15))
    im = ax[0].imshow(image_power(img1), norm=LogNorm(vmin=5))
    im = ax[1].imshow(image_power(img2), norm=LogNorm(vmin=5))
    plt.show()
    plt.show()

plot_2d_fft(cat_grayscale, cat_grayscale_equalized)
```



The interpretation of the 2-dimensional power spectrum plot requires some explanation. The plot shows both vertical and horizontal frequency components. The Fourier transform values can be positive and negative, and symmetric for the real-valued image. The square of the absolute value of these components are the same. Therefore the values in the four quadrants of the plot will be identical.

The zero frequency (constant) component of the image are is at the center. There zero vertical frequency components can be seen as a centered horizontal line and the zero horizontal frequency components can be seen as a vertical line. These lines divide the four identical quadrants of the plot.

The further from the center a component is the higher the frequency. The maximum (at the Nyquist rate) vertical and horizontal frequency component is in the corner of the plot. Intermediate frequencies are between the center and the corner.

Compare these displays. First, notice that much of the power (energy) in the image is at or near 0 frequency, indicating that the values across the image are mostly constant. Second, you can see that equalization has amplified the higher frequency components for both dimensions of the image.

1.1 Convolution in 1 dimension

To get started, let's work with the simplest case of one dimensional convolution. Convolution is a process of applying a filter of a certain span over one of more dimensions of the data.

To conceptually understand 1-d convolution examine Figure 1.1 below.. At each time step a value for output series is computed by applying the convolution operator to the input series. The convolution operator is a set of weights $\{W_1, W_2, W_3\}$ which are mutilied by the input values. The weighted inputs are then summed to compute the value of the output series. The operator is then moved along the input series (by the stride of 1 in this case). The procecss proceeds until the end of the series is reached.

Notice that the output series is shorter than the input series by $span - 1$. In the illustrated example the kernel has a span of 3, makig the output series length by 2. This reduction in the length of the output is inherent in all convolution operations.

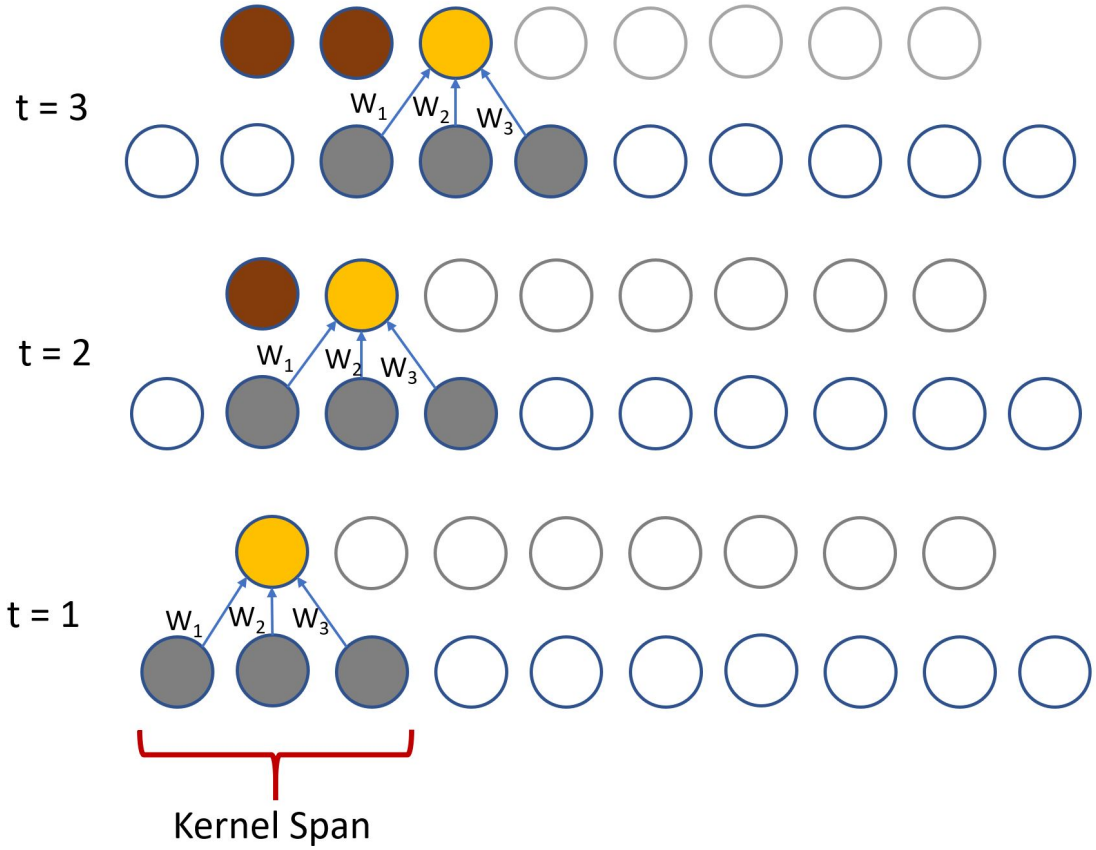


Figure 1.1. 1-d convolution with operator span of 3

Mathematically, we write the convolution operation:

$$s(t) = (x * k)(t) = \sum_{\tau=-\infty}^{\infty} x(t)k(t-\tau)$$

where,

$s(t)$ is the output of the convolution operation at time t,

x is the series of values,

k is the convolution kernel,

$*$ is the convolution operator,

τ is the time difference of the convolution operator.

Summing over the range $\{-\infty, \infty\}$ seems to be a problem. However, in practice the convolutional kernal has a finite span, s. For an operator with an odd numbered span, we can then rewrite the above equation as:

$$s(t) = (x * k)(t) = \sum_{\tau=t-a}^{t+a} x(\tau)k(t-\tau)$$

Where, $a = \frac{1}{2}(kernel\_span - 1)$, for an odd length kernel.

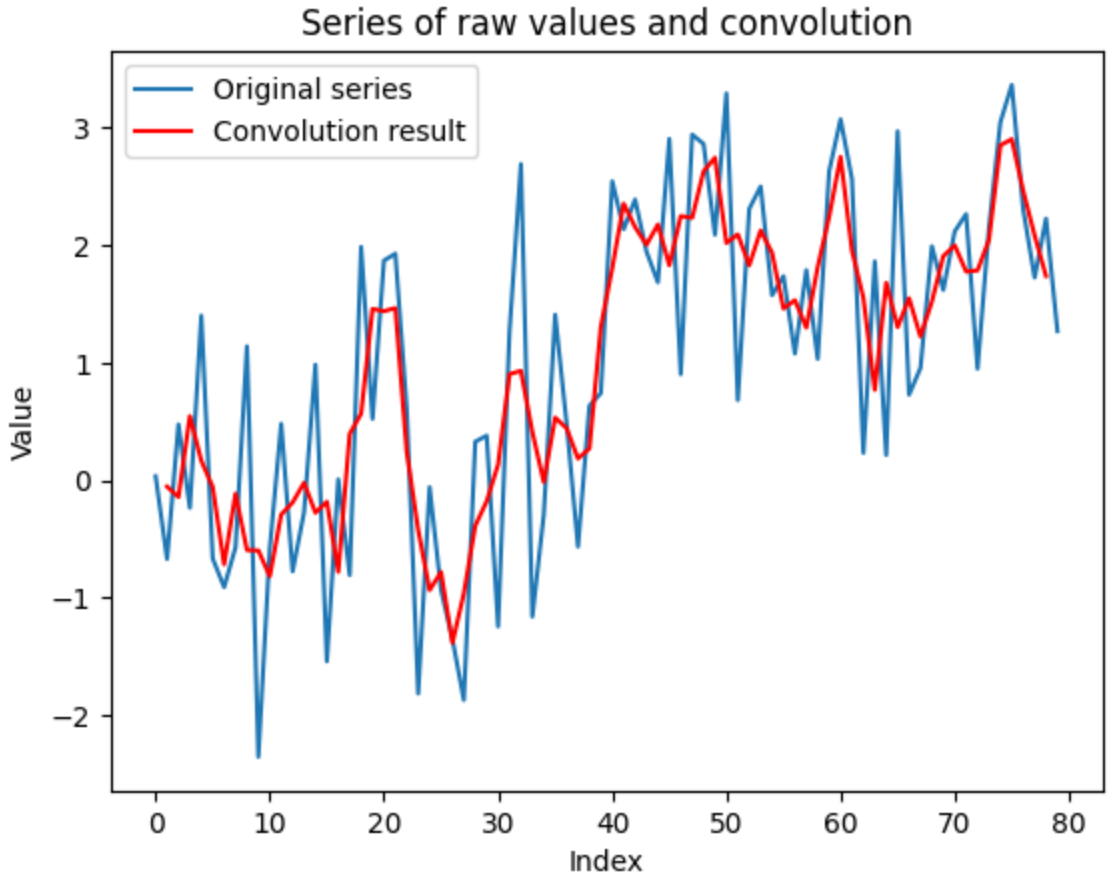
In other words, we only need to take the sum over the span of the convolution operator. Notice that this operator is non-causal, since values from ahead of the time step being operated on are used to compute the output.

> **Exercise 2-2:** To make this all a bit more concrete, you will try a code example. You will construct a short filter and apply it over a 1-d series. In the cell below you will implement a simple 1-d convolution operation given a set of kernel weights. The core of this operation is a dot product between the subset of values of the series and the kernel weights. You will first use a 3-point moving average kernel, $[0.333, 0.333, 0.333]$. Do the following:
>
> 1. Create a function that performs convolution of the moving average kernel with the random series. You can perform this operation by iteratively calling numpy.dot as you move the operator along overlapping 3-point segments of the series. Your function should have two arguments, the series and the kernel. the Make sure you account for the fact that for this operator with span of 3 (an odd number) the length of the result will be 2 points shorter than the original series, one point shorter on each end.
> 2. Use the code provided to create a step-function series with random noise added.
> 3. Execute your function and plot the result using the function provided.
> 4. Finally, execute your function and plot the result using a 5-point Gaussian kernel $[0.05, 0.242, 0.399, 0.242, 0.05]$.

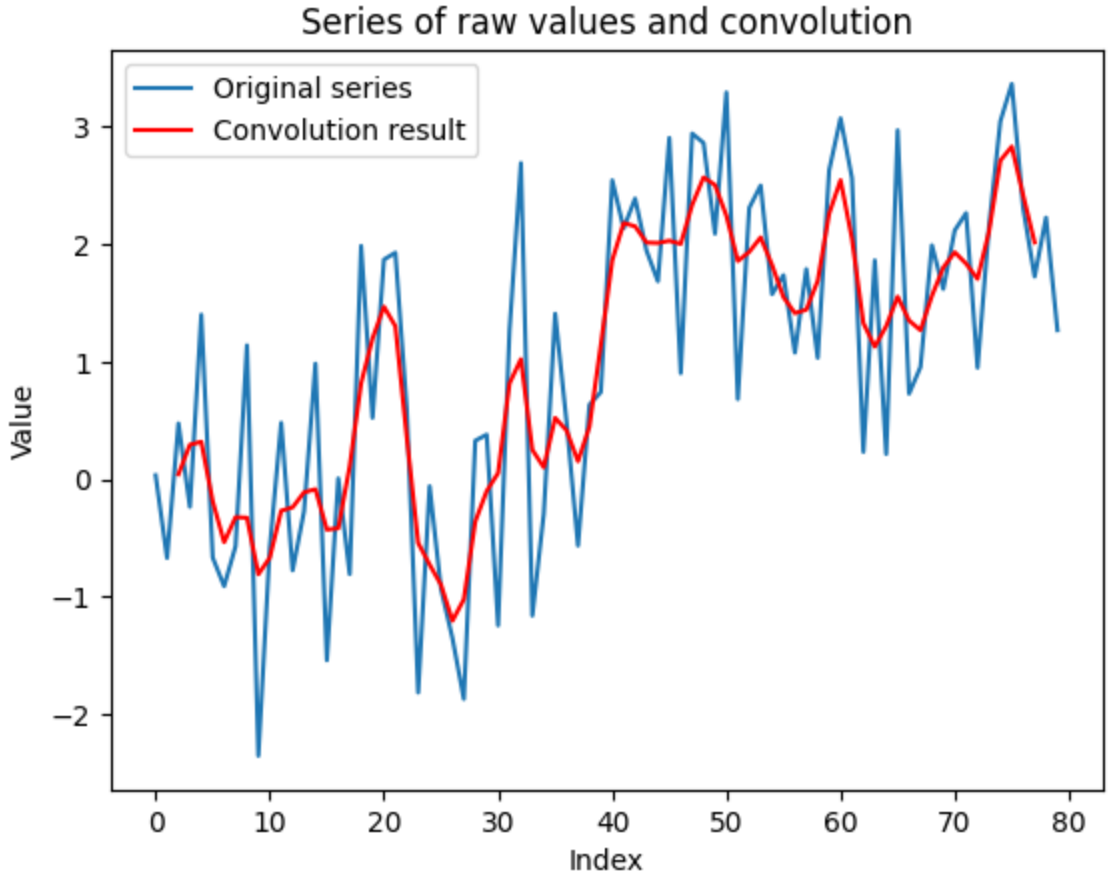
In [ ]:
```
def plot_conv(series, conv, span):
    x = list(range(series.shape[0]))
    offset = int((span-1)/2)
    end = series.shape[0] - offset
    plt.plot(x, series, label = 'Original series')
    plt.plot(x[offset:end], conv, color = 'red', label = 'Convolution result
    plt.legend()
    plt.xlabel('Index')
    plt.ylabel('Value')
    plt.title('Series of raw values and convolution')
```

```
    plt.show()

np.random.seed(12233)
series = np.concatenate((np.random.normal(size = 40), np.random.normal(size


## Complete the function below
def conv_1d(series, kernel):
    '''Performs simple 1d convolution over the series values
    given the kernel weights. This function only works for
    kernels with odd span'''

    results = []

    a = len(kernel) - 1
    [results.append(np.dot(kernel, series[i : i + len(kernel)])) for i in ra

    return results


kernel1 = np.array([0.333, 0.333, 0.333])
result1 = conv_1d(series, kernel1)
print('Length of result1 = ' + str(len(result1)))
plot_conv(series, result1, 3)

kernel2 = np.array([0.05, 0.242, 0.399, 0.242, 0.05])
result2 = conv_1d(series, kernel2)
print('Length of result2 = ' + str(len(result2)))
plot_conv(series, result2, 5)
```

```
Length of result1 = 78
```




```
Length of result2 = 76
```



> Answer the following questions:
>
> 1. What effect can you observe when comparing the original and convolved series with respect to both overall mean and noise spikes?
> 2. What reason(s) can you think of for the differences in the convolved series?
> 3. Are both of these kernels properly normalized and why?
>
> **End of exercise.**

> **Answers:**
>
> 1. the peaks are attenuated while maintaining the overall mean.
>
> 2. results2 is smoother than results1 because the Gaussian kernel has a larger span and therefore more influence on the output. That makes results2 to lose detail compared to results1.
>
> 3. Considering that the sum of the kernels is not exactly 1, we could say they are not properly normalized. However, the values used are pretty close, which makes the difference almost negligible.

Applying 2-d Convolution

Next, let's look at how we can apply convolution operations in two dimensions. The theory is the same as the 1d case.

Figure 1.2 illustrates a simple example of applying a convolution operator to a 2-d input array. The 2-d input array can be a gray-scale image, for example.

A 3X3 convolutional operator is illustrated in Figure 1.2. This operator is applied to a small image of dimension of 4X4 pixels. Within the 4X4 image the operator is only applied four times, where the complete operator lies on the image. Convolution where the operator is completely contained within the input sample is sometimes called **valid convolution**.

In this case the 4X4 pixel input results in a 2X2 pixel output. You can think of the convolutional operator mapping from the center of the operator input to pixel in the output. Similar to the 1-d case, for a convolution operator with span s in both dimensions the output array will be reduced in each dimension by $\frac{s+1}{2}$

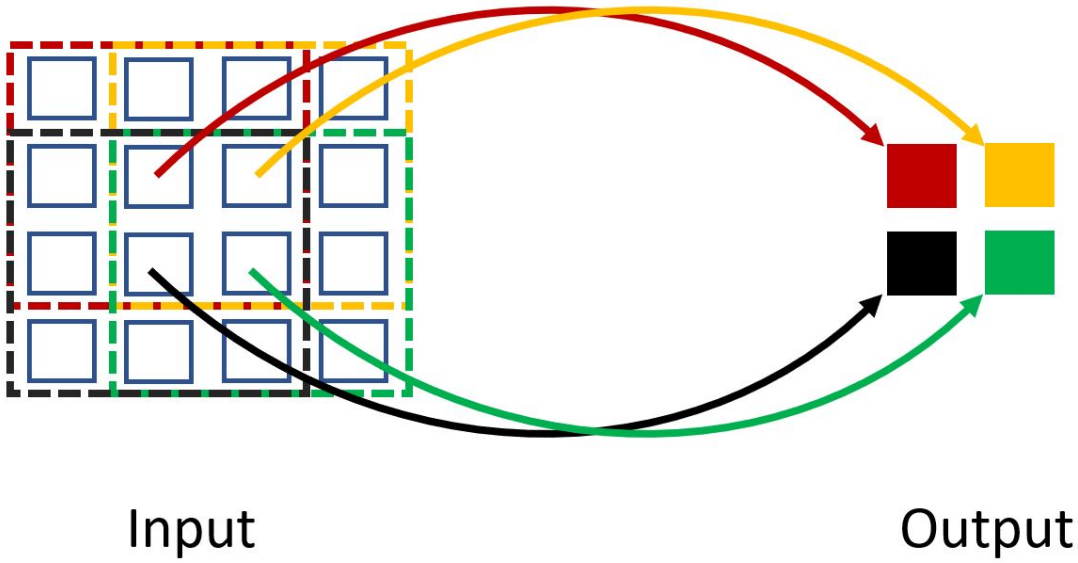


Figure 1.2. Simple 2-d convolution operation

For a convolution with the identical span s in both dimensions we can write the convolution operations as:

$$S(i,j) = (I * K)(i,j) = \sum_{m=i-a}^{i+a} \sum_{n=j-a}^{j+a} I(i,j)K(i-m, j-n)$$

Notice that S, I and K are all tensors. Given this fact, the above formula is easily extended to higher dimensional problems.

The convolutional kernel K and the image I are comutitive so there is an alternative representation by applying an operation known as **kernel flipping**. Flipped kernel convolution can be represented as follows:

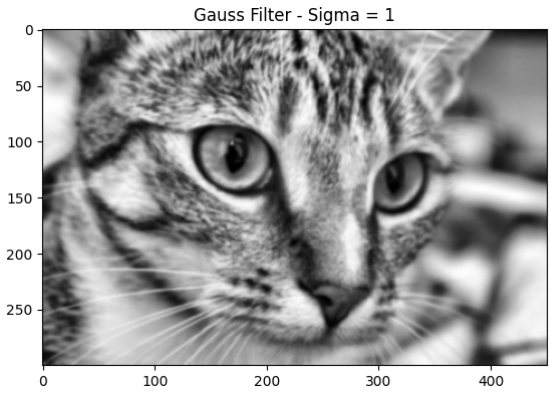
$$s(i,j) = (I * K)(i,j) = \sum_{m=i-a}^{i+a} \sum_{n=j-a}^{j+a} I(i-m, j-n)K(i,j)$$

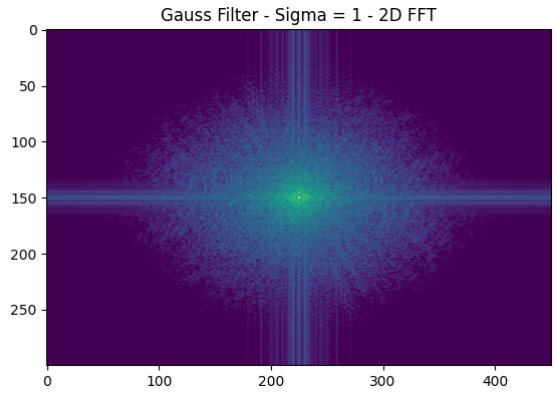
Gaussian Filtering and Noise Reduction

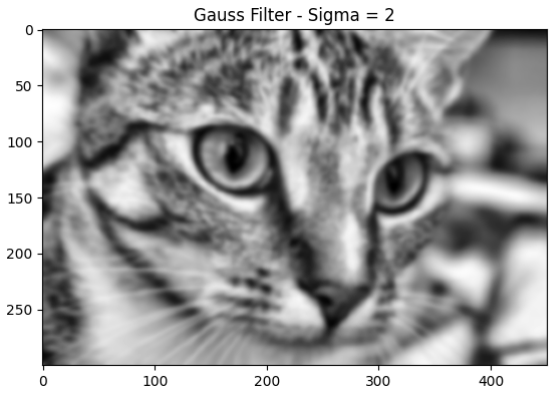
A Gaussian filter computes a 2-dimensional weighted moving average as it is passed across the image. The weights are determined by a 2-dimensional truncated Gaussian function. The result is a lower-bandwidth or smoothed version of the image. The span of the Gaussian function is determined by the standard deviation, σ, of the Gaussian. The wider the span or the lower the bandwidth the more smoothing of the filtered image.

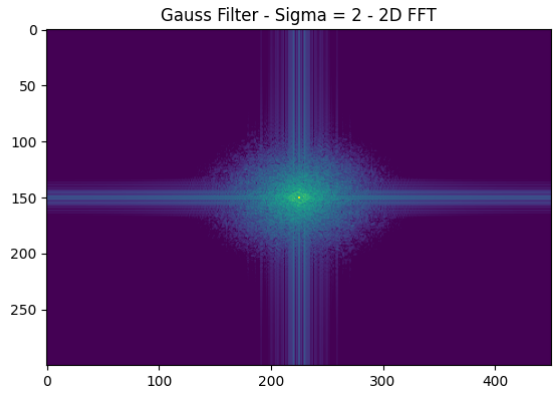
> **Exercise 2-3:** The wider the span of the Gaussian filter the narrower the bandwidth, which has the effect of blurring the image. To see this effect with Gaussian filtered images of the gray-scale equalized cat image plot the filtered image in 4 rows of a plot array using values of $\sigma = [1.0, 2.0, 3.0, 4.0]$. Use the skimage.filters.gaussian function. Make sure the titles of the images indicate the value of sigma. In the other column of each row display the 2-dimensional FFT of the filtered image.

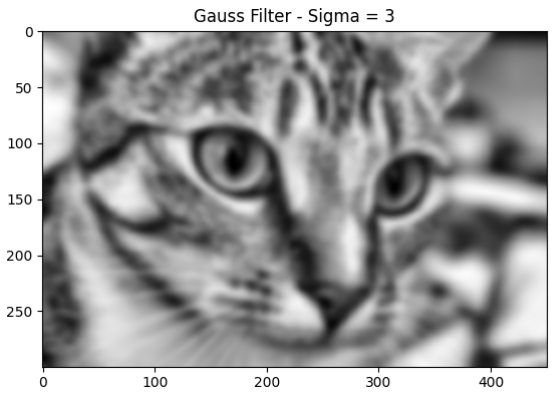
In [ ]:
```
fig, ax = plt.subplots(4, 2, figsize = (15, 20))
ax = ax.flatten()

## Complete the code below
for i, sigma in enumerate([1.0, 2.0, 3.0, 4.0]):
    cat_gaussian = skfilters.gaussian(cat_grayscale_equalized, sigma = sigma
    ax[i * 2].set_title(f"Gauss Filter - Sigma = {i + 1}")
    ax[i * 2].imshow(cat_gaussian, cmap = 'gray')
    ax[i * 2 + 1].set_title(f"Gauss Filter - Sigma = {i + 1} - 2D FFT")
    ax[i * 2 + 1].imshow(image_power(cat_gaussian), norm = LogNorm(vmin = 5)

plt.show()
```

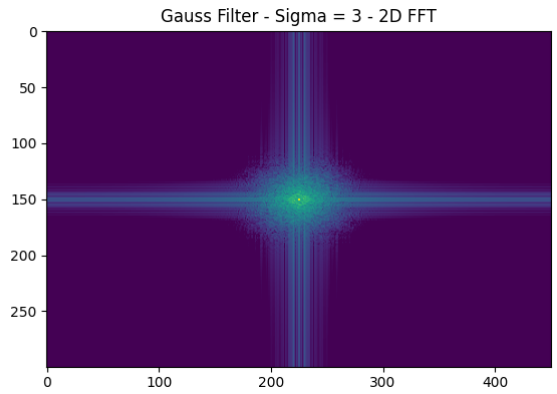


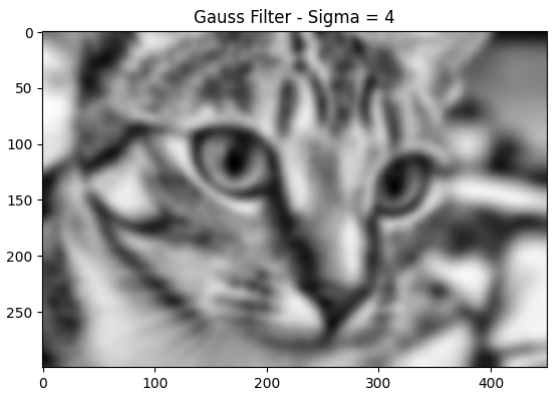


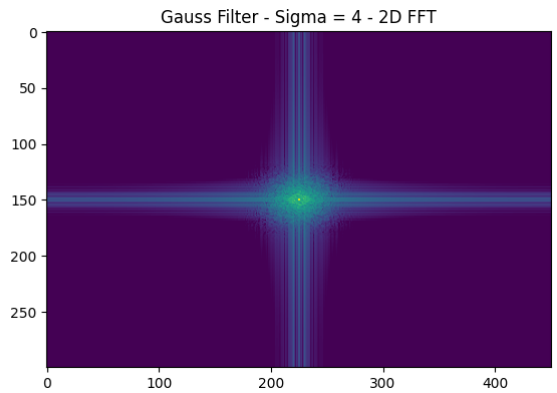












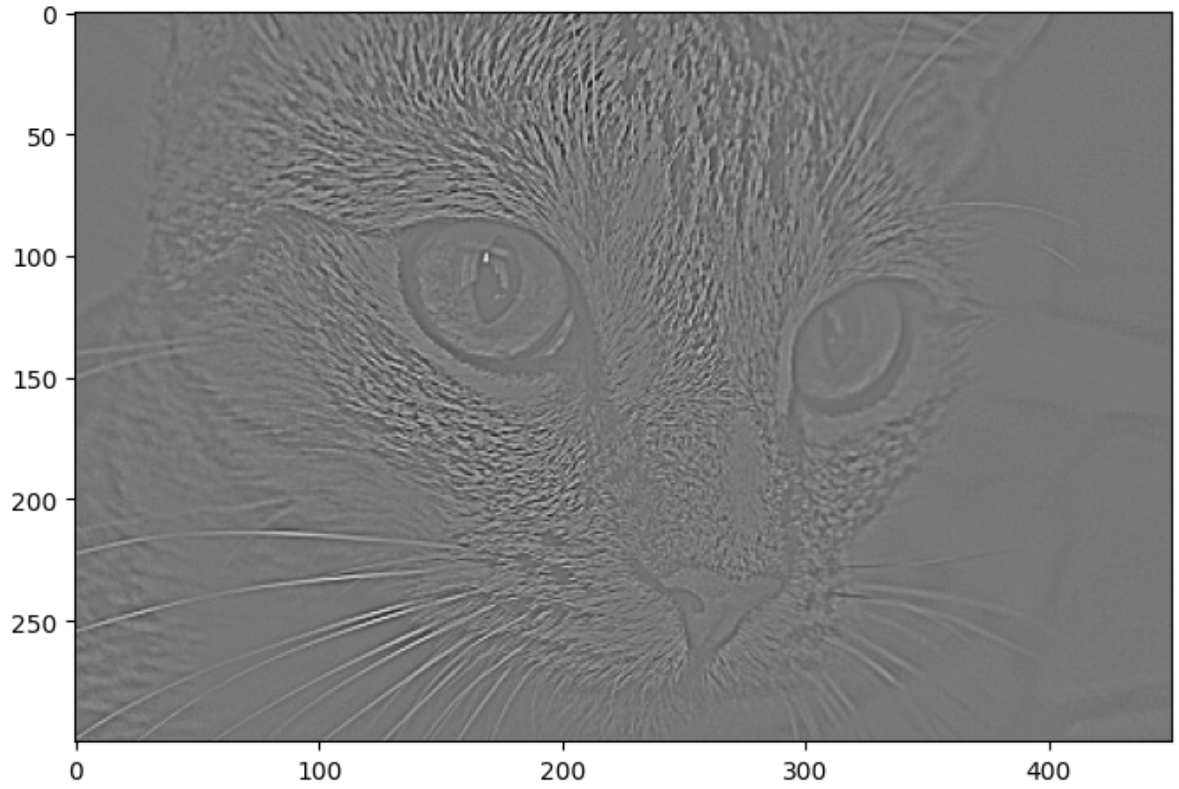
> 1. How does the blurring of the image and the bandwidth shown by the FFT change with the value of σ?
> 2. Compare the FFTs of the filtered image to the original equalized image. How has the bandwidth of the image features changed with the value of σ?

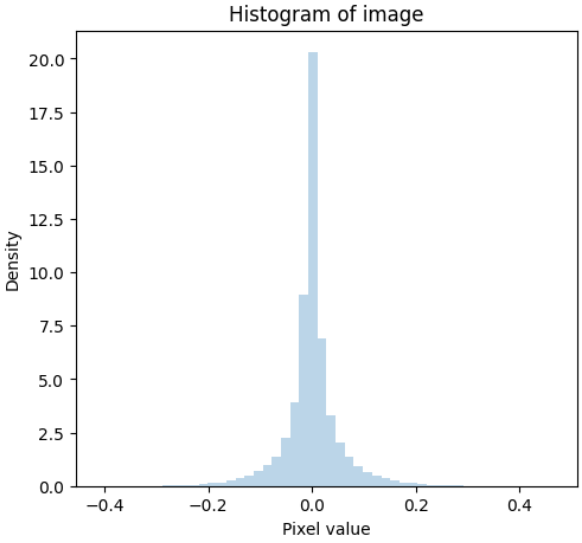
> **End of exercise.**

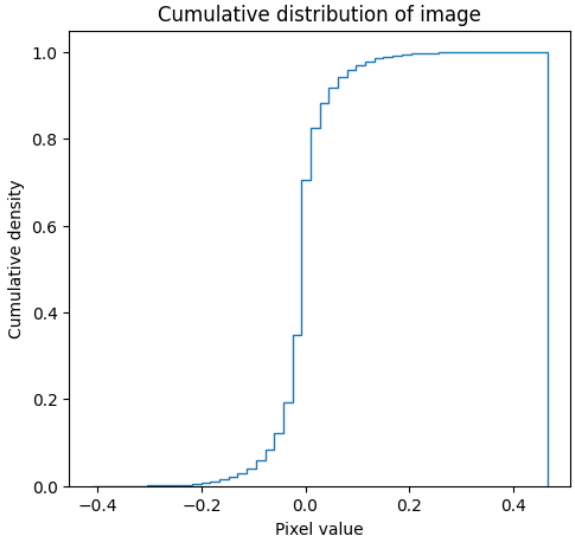
> **Answers:**
>
> 1. The blurring of the image increases with the value of σ. The bandwidth shown by the FFT decreases with as σ increases, as shown on the right column.
>
> 2. The bandwidth of the image decreases as the value of σ increases. This is because the Gaussian filter is a low-pass filter, which means it attenuates high frequencies.

> **Exercise 2-4:** Examining the difference between the original image and filtered image can help you understand the effect of the filter. To create this display, do the following:
>
> 1. Compute the Gaussian filtered image with $\sigma = 1.0$ from the gray-scale equalized cat image.
> 2. Compute the difference between the original gray-scale equalized image and the Gaussian filtered image.
> 3. Display the image of the difference and the distribution of the difference.

In [ ]:
```
## Put your code below

# 1. Compute the Gaussian filtered image with sigma = 1.0 from the
#    gray-scale equalized cat image.
cat_gaussian = skfilters.gaussian(cat_grayscale_equalized, sigma = 1.0)

# 2. Compute the difference between the original gray-scale equalized
#    image and the Gaussian filtered image.
cat_difference = cat_grayscale_equalized - cat_gaussian

# 3. Display the image of the difference and the distribution of the
#    difference.
plot_grayscale(cat_difference)
plot_gray_scale_distribution(cat_difference)
```







> Examine the plots and answer the following questions:
>
> 1. What aspects of the cat image has the difference between the original and filtered images have been highlighted?
> 2. What does the narrow range of pixel values of the difference tell you about the differences between the original and filter images?
>
> **End of exercise.**

> **Answers:**

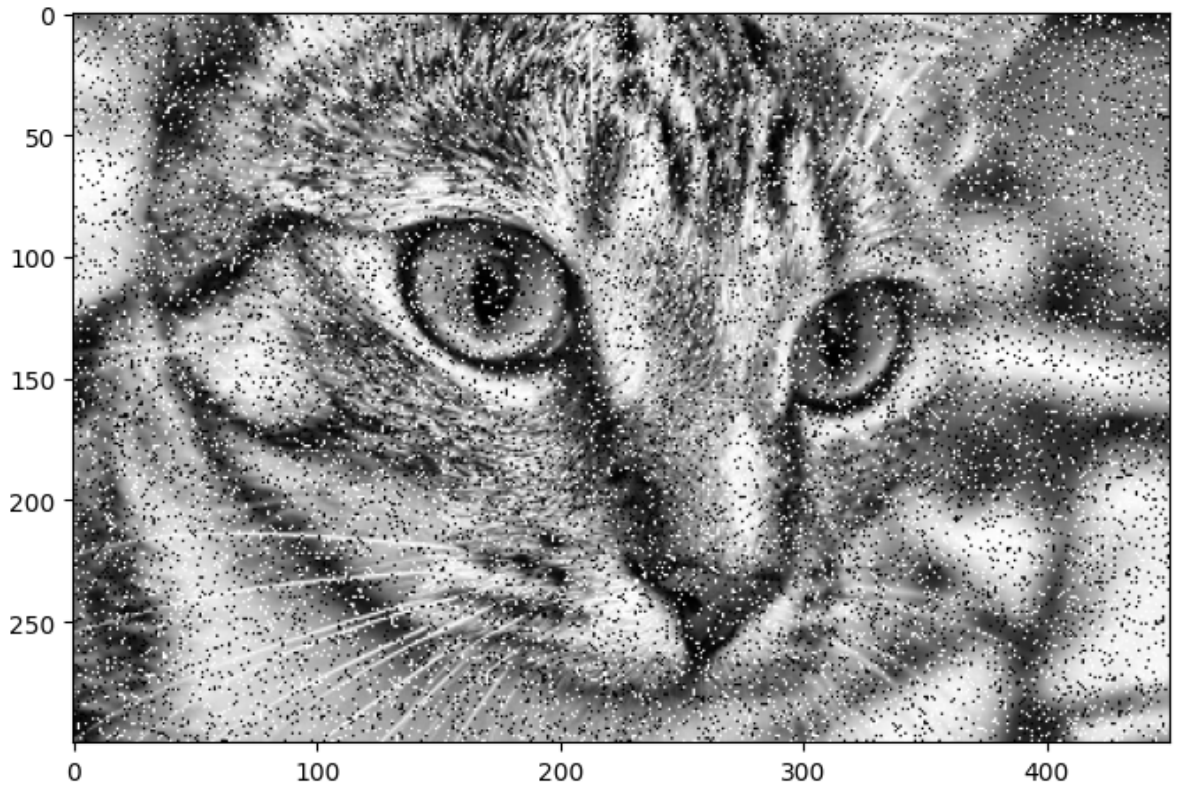
> 1. Given that the Gaussian filter is a low-pass filter, it attenuates high frequencies highlighting the edges of the cat image.
>
> 2. The narrow range of the difference indicates that the original image and the filtered image are very similar.

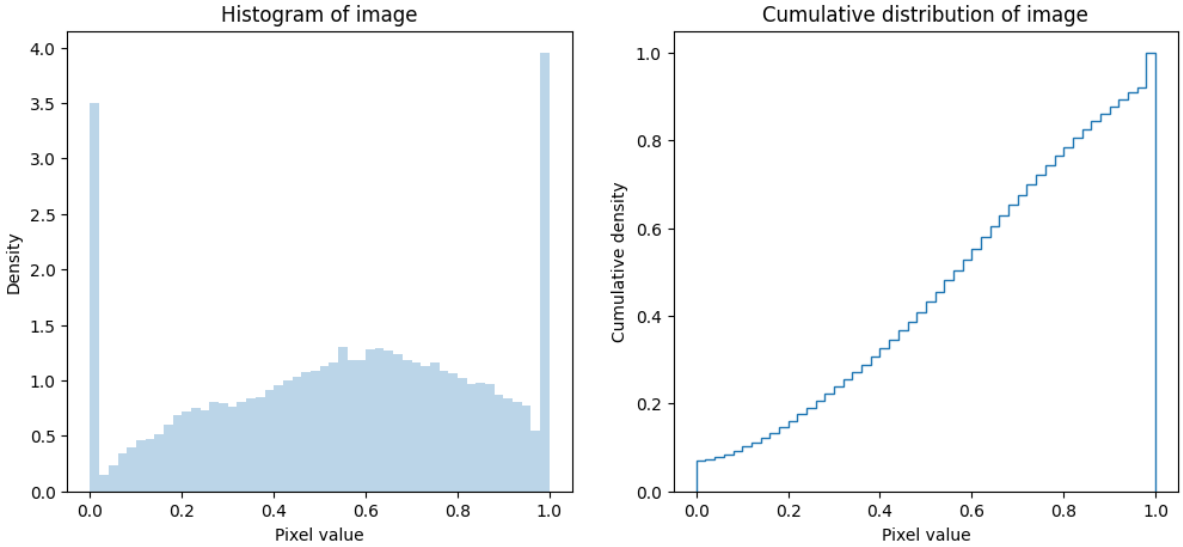
Gaussian Filtering with Shot Noise

One of the primary purposes of applying filtering to images is the suppression of noise. Many computer vision algorithms, particularly machine learning algorithms, have poor noise robustness. We will now investigate the use of filters for noise suppression.

A simple model of image noise is know as **shot noise** or **salt and pepper noise**. In this model, the noise is comprised of random pixel values at the extreme of the range. For example for unsigned integer values the noise has values of $[0, 255]$ or for floating point values, $[0.0, 1.0]$. To see an example of salt and pepper noise on the cat image execute the code in the cell below.

In [ ]:
```
cat_grayscale_equalized_noise = np.copy(cat_grayscale_equalized).flatten()
cat_grayscale_equalized_noise[np.random.choice(cat_grayscale_equalized_noise
cat_grayscale_equalized_noise[np.random.choice(cat_grayscale_equalized_noise
cat_grayscale_equalized_noise = cat_grayscale_equalized_noise.reshape(cat_gr

plot_grayscale(cat_grayscale_equalized_noise)
plot_gray_scale_distribution(cat_grayscale_equalized_noise)
```

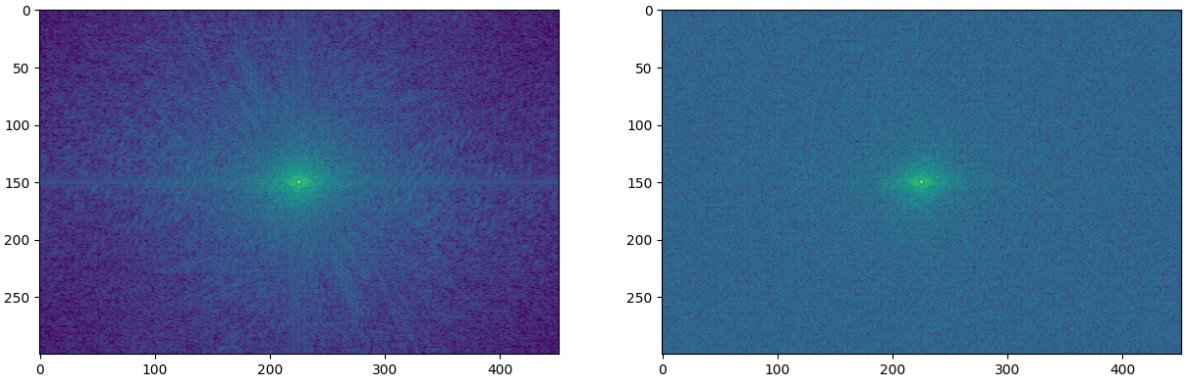




Examine the image noticing the light pixels with value of 1.0 and the dark pixes with value of 0.0 randomly spread throughout the image. Further, the distribution of pixel values show spikes at 0.0 and 1.0.

Execute the code in the cell below to display the FFTs of the equalized image and the image with the salt and pepper noise.

In [ ]:
```
plot_2d_fft(cat_grayscale_equalized, cat_grayscale_equalized_noise)
```



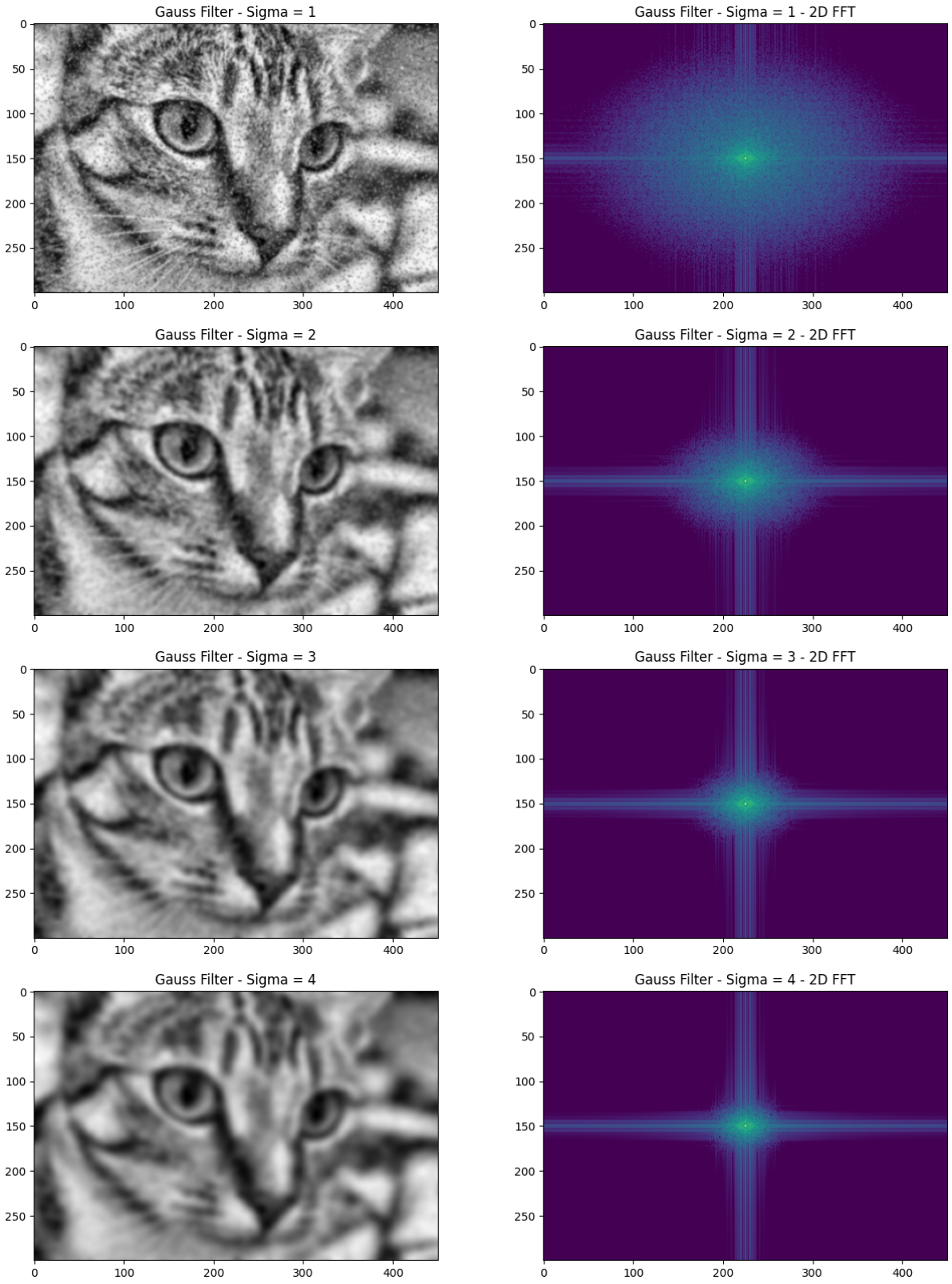
The image with the salt and pepper noise added has a much wider bandwidth. Further, the higher frequency components of the FFT of the salt and pepper noise image appears nearly uniform random.

We will now explore how filters might be able to improve the quality of this noisy image.

> **Exercise 2-5:** You will now investigate how the noise added to the gray-scale cat image can be suppressed using a Gaussian filter. To changing the Gaussian filter span effects the image. To see this effect plot the Gaussian filtered image in 4 rows of a plot array using values of $\sigma = [1.0, 2.0, 3.0, 4.0]$. Make sure the titles of the images indicate the

> value of sigma. In the second column of each row display the 2-dimensional FFT of the filtered image.

In [ ]:
```
fig, ax = plt.subplots(4, 2, figsize=(15, 20))
ax = ax.flatten()

## Complete the code below
for i, sigma in enumerate([1.0, 2.0, 3.0, 4.0]):
    cat_gaussian_filtered = skfilters.gaussian(cat_grayscale_equalized_noise
    ax[i * 2].set_title(f"Gauss Filter - Sigma = {i + 1}")
    ax[i * 2].imshow(cat_gaussian_filtered, cmap = 'gray')
    ax[i * 2 + 1].set_title(f"Gauss Filter - Sigma = {i + 1} - 2D FFT")
    ax[i * 2 + 1].imshow(image_power(cat_gaussian_filtered), norm = LogNorm(

plt.show()
```



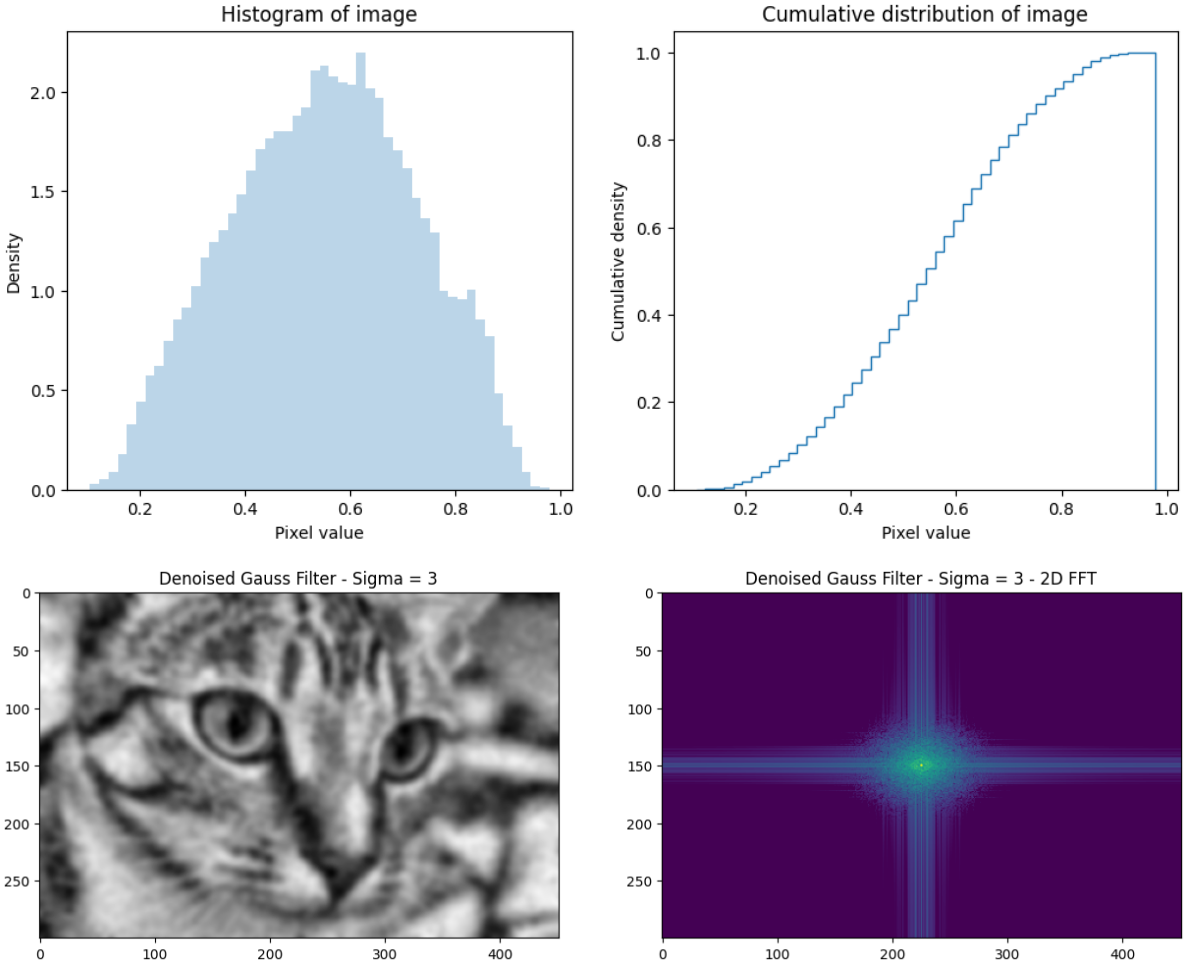
> Examine the images. How can you describe the trade-off between the span (or bandwidth) of the Gaussian filter, the suppression of the salt and pepper noise, and the resolution of the image?
> **End of exercise.**

> **Answer:**

> This is an interesting trade-off where the span of the Gaussian filter is inversely proportional to the suppression of the salt and pepper noise. The resolution of the image is also inversely proportional to the span of the Gaussian filter.

> **Exercise 2-6:** Somewhat subjectively, the image with $\sigma = 3$ is a reasonable trade off between blurring and noise suppression. The next question to address is what the filtering has done to the distribution of pixel values and the bandwidth of the image. To find out do the following:
>
> 1. Compute a filtered version of the noisy gray-scale cat image with $\sigma = 3$
> 2. Plot the distribution of the pixel values of the filtered image.
> 3. Display the FFT of the noisy image next to the filtered image.

In [ ]:
```
## Put your code below

# 1. Compute a filtered version of the noisy gray-scale cat image with sigma
cat_gaussian_filtered = skfilters.gaussian(cat_grayscale_equalized_noise, si

# 2. Plot the distribution of the pixel values of the filtered image
plot_gray_scale_distribution(cat_gaussian_filtered)

# 3. Display the FFT of the noisy image next to the filtered image.
fig, ax = plt.subplots(1, 2, figsize=(15, 20))
ax = ax.flatten()
ax[0].set_title(f"Denoised Gauss Filter - Sigma = 3")
ax[0].imshow(cat_gaussian_filtered, cmap = 'gray')
ax[1].set_title(f"Denoised Gauss Filter - Sigma = 3 - 2D FFT")
ax[1].imshow(image_power(cat_gaussian_filtered), norm = LogNorm(vmin = 5))

plt.show()
```



> Examine the plots and answer the following questions:
>
> 1. When comparing the distribution of pixel values of the filtered image to the noisy image what change do you see? Given the properties of the Gaussian filter, how can you explain this change?
> 2. Noting the differences in the FFTs of the filtered and noisy images, what has the filter done to the frequency components of the image? Given the properties of the Gaussian filter, how can you explain this change?
>
> **End of exercise.**

> **Answers:**

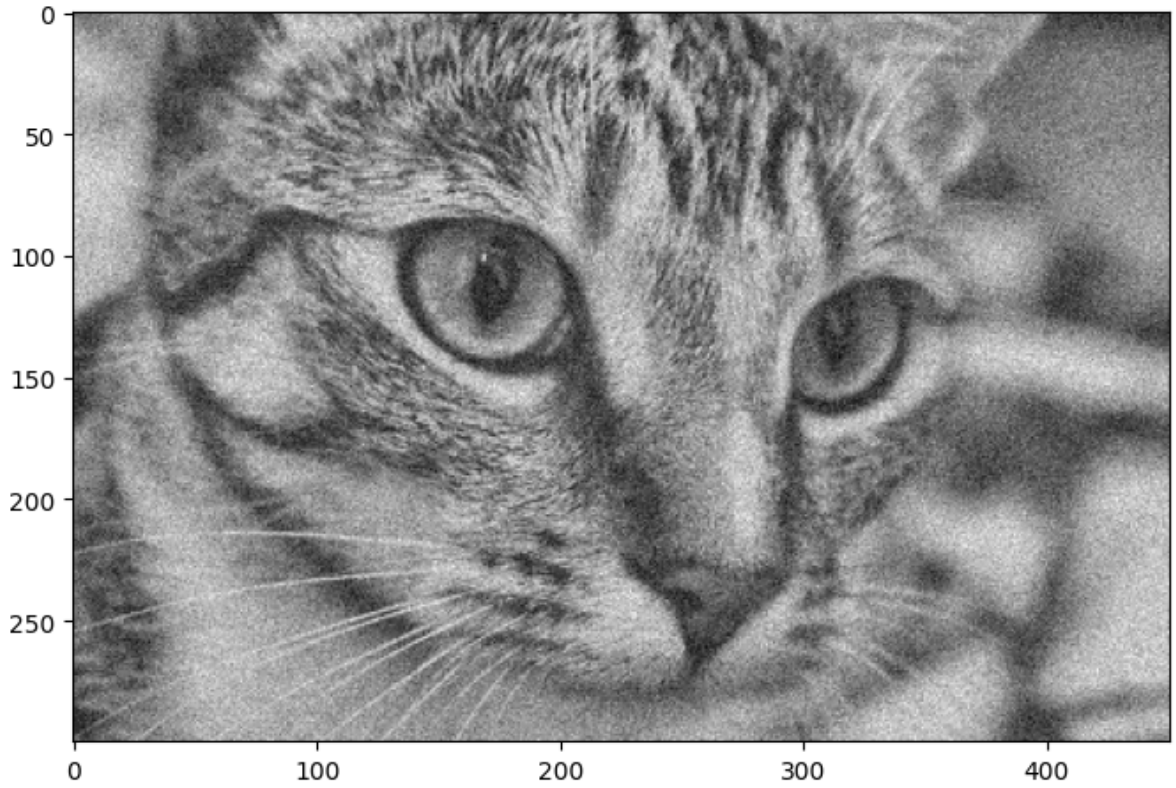
> 1. I see two main differences: First, the spikes at 0.0 and 1.0 dissapeared. Second, the distribution of pixel values of the filtered image is narrower and more dense in the mid values than the noisy image. As a result, the filtered image looks better.

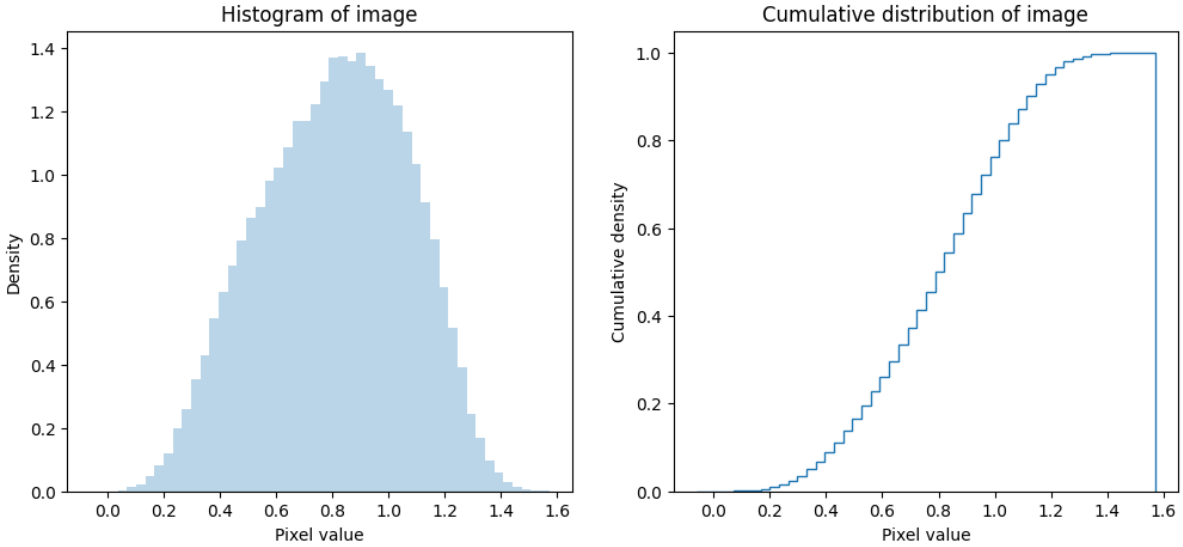
> 2. The Gaussian filter attenuated the high frequencies blurring the image. The way the Gaussian filter attenuates high frequencies is by applying convolution to neighboring pixel values reducing peaks. This is how the filtered image looks better although it is blurred.

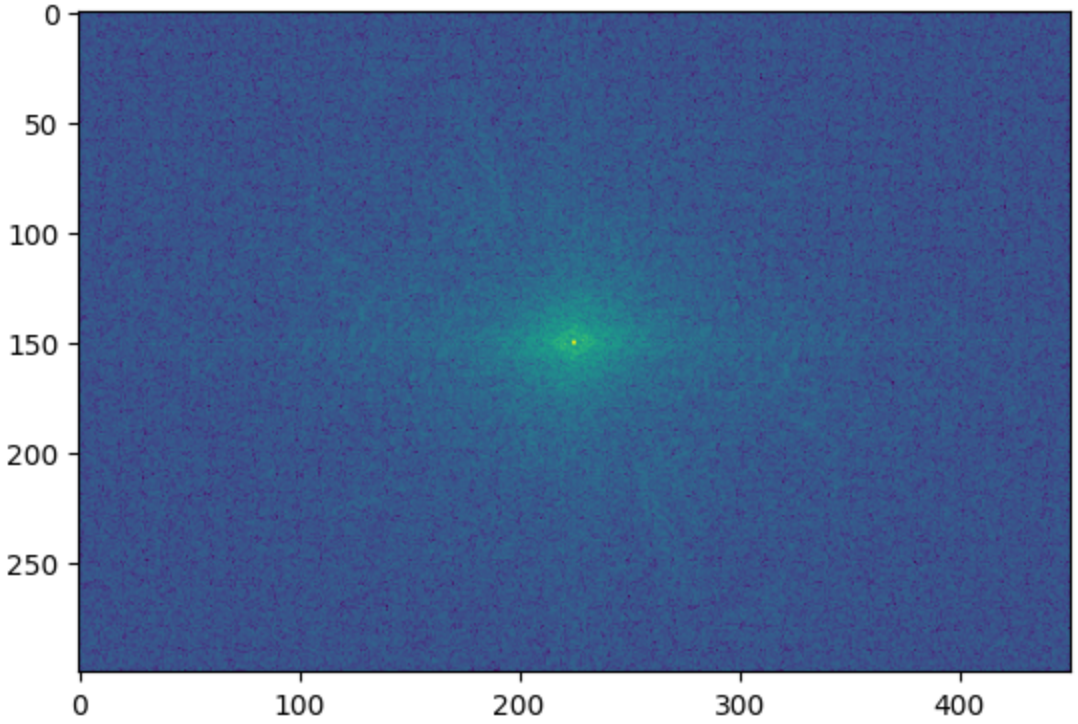
Gaussian Filtering with Gaussian Noise

We will now add Gaussian noise to the cat image. Execute the code in the cell below and examine the results.

In [ ]:
```
sigma = 0.1
cat_grayscale_equalized_gaussian = np.copy(cat_grayscale_equalized).flatten(
cat_grayscale_equalized_gaussian = cat_grayscale_equalized_gaussian + np.rar
cat_grayscale_equalized_gaussian = cat_grayscale_equalized_gaussian.reshape(

plot_grayscale(cat_grayscale_equalized_gaussian)
plot_gray_scale_distribution(cat_grayscale_equalized_gaussian)
_ = plt.imshow(np.square(np.abs(fft.fftshift(fft.fft2(cat_grayscale_equalize
```





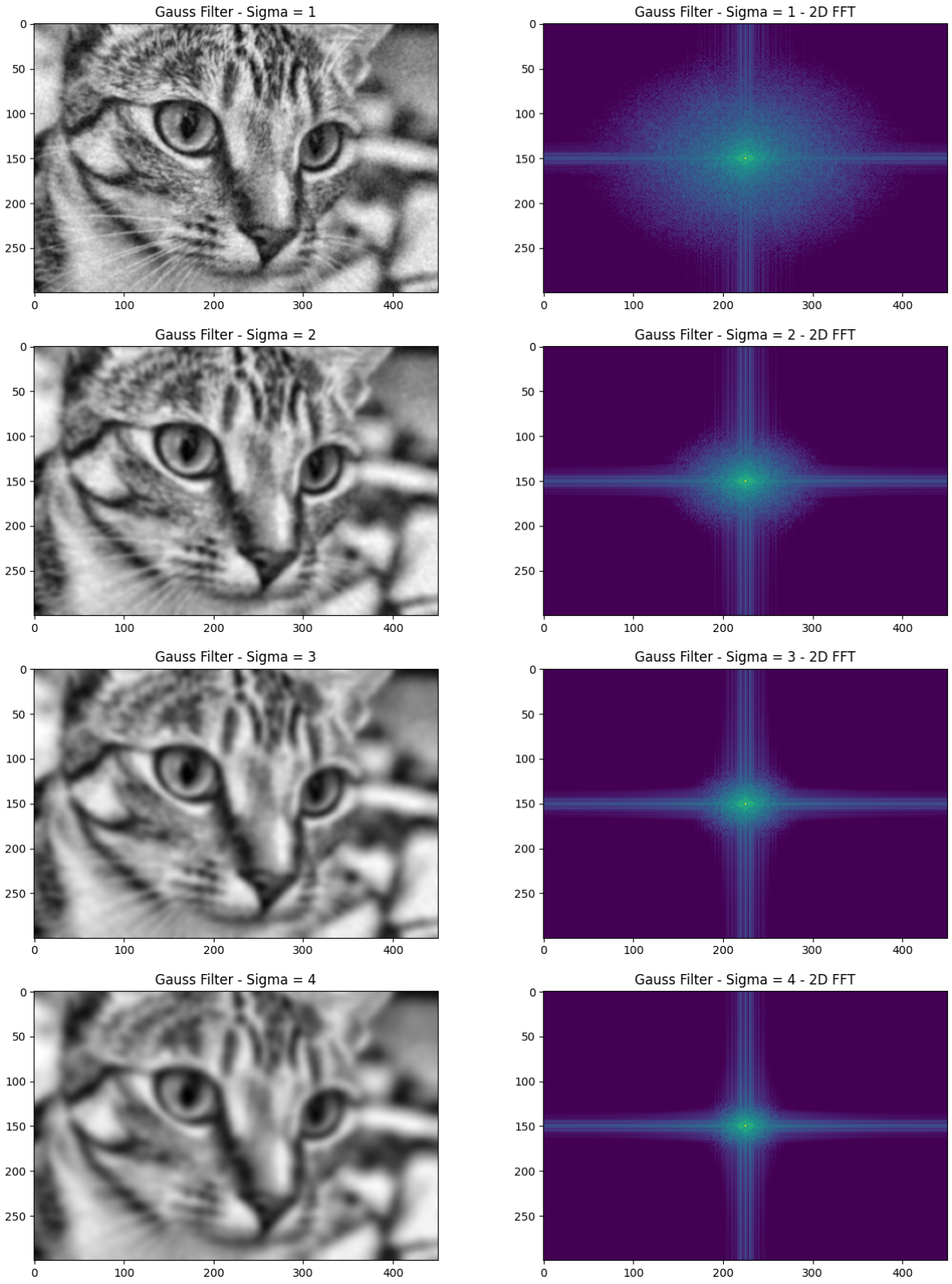


The Gaussian noise is evident in the image. Notice how the Gaussian noise has changed the distribution of the pixel values and results in a broad spectrum.

For this example, we will ignore the fact that the pixel values are out to the usual range.

> **Exercise 2-7:** You will now apply Gaussian filters to the gray-scale cat image with Gaussian noise. The filters will have σ values of $[1.0, 2.0, 3.0, 4.0]$. Make sure the titles of the images indicate the bandwidth value, σ. In the second column of each row display the 2-dimensional FFT of the filtered image.

In [ ]:
```
fig, ax = plt.subplots(4, 2, figsize=(15, 20))
ax = ax.flatten()


## Put your code below
for i,sigma in enumerate([1.0, 2.0, 3.0, 4.0]):
    cat_gaussian_filtered = skfilters.gaussian(cat_grayscale_equalized_gauss
    ax[i * 2].set_title(f"Gauss Filter - Sigma = {i + 1}")
    ax[i * 2].imshow(cat_gaussian_filtered, cmap = 'gray')
    ax[i * 2 + 1].set_title(f"Gauss Filter - Sigma = {i + 1} - 2D FFT")
    ax[i * 2 + 1].imshow(image_power(cat_gaussian_filtered), norm = LogNorm(

plt.show()
```



1. How does the blurring of the image and the bandwidth shown by the FFT change with the bandwidth of the Gaussian filter?
2. Compare the FFTs of the filtered image to the original equalized image. How has the bandwidth of the image features changed?

End of exercise.

> **Answer:**
>
> 1. As in previous answers, the blurring of the image increases with the value of σ. The bandwidth shown by the FFT decreases as σ increases, as shown on the 2D FFT.

> 2. High frequencies were attenuated in the filtered image, which is what caused the blurring.

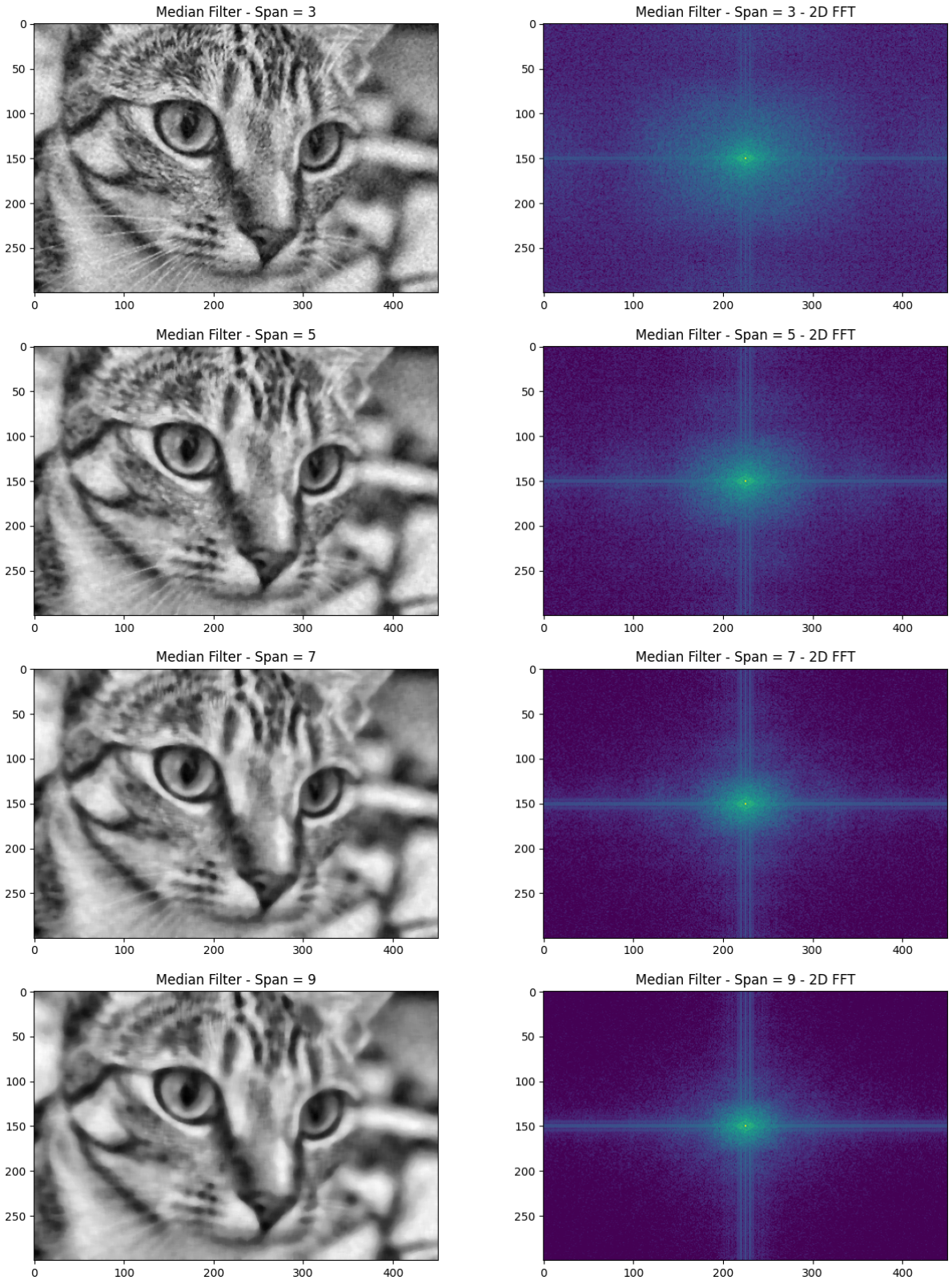
Median Filtering

You have now explored the basic properties of Gaussian filters. Now we will turn our attention to the median filter which has noticeably different properties. The median filter is simple in principle. The filter replaces the pixel value at the center of the filter operator with the median value over the filter area.

As with the Gaussian filter the span of the median filter must be selected. This span is almost always an odd number so there is a central pixel. The greater the span the lower the bandwidth of the filter. Lower bandwidth has the effect of reducing noise, but increasing the blurring effect.

> **Exercise 2-8:** You will now explore how changing the median filter span effects the image. To see this effect, start with the gray-scale cat image with added Gaussian noise and then plot the median filtered image in 4 rows of a plot array using values of span $= [3, 5, 7, 9]$. Make sure the titles of the images indicate the span value. In the second column of each row display the 2-dimensional FFT of the filtered image.

In [ ]:
```
fig, ax = plt.subplots(4, 2, figsize = (15, 20))
ax = ax.flatten()

## Put your code below
for i, width in enumerate([3, 5, 7, 9]):
    cat_gaussian_filtered = skfilters.median(cat_grayscale_equalized_gaussia
    ax[i * 2].set_title(f"Median Filter - Span = {width}")
    ax[i * 2].imshow(cat_gaussian_filtered, cmap = 'gray')
    ax[i * 2 + 1].set_title(f"Median Filter - Span = {width} - 2D FFT")
    ax[i * 2 + 1].imshow(image_power(cat_gaussian_filtered), norm = LogNorm(

plt.show()
```



1. How does the blurring of the image and the bandwidth shown by the FFT change with the span of the median operator?
2. Compare the FFTs of the filtered image to the original equalized image. How has the bandwidth of the image features changed?

End of exercise.

> **Answer:**

> 1. The median filter is less aggresive than the gaussian filter, so the blurring is less. The bandwidth shown by the FFT is less concentrated in the center, leaving some high frequencies on the horizontal and vertical axes of the FFT.

> 2. The FFT of the original equalized is more concentrated in the center, while the FFT of the median filtered image is more spread out with a notable redduction in the center of each quadrant that expands to the edges of the FFT.

Convolution with Separable Kernels

We now turn our attention to efficient computation of 2-dimensional convolutions. If we directly applied a $k \times k$ convolutional operator, each point computed required k^2 multiplication and addition operations. Is there a way we can make this operation more efficient? Yes! Use a **separable kernel**.

Most linear convolutional operators can be separated into two 1-dimensional kernels, one horizontal and one vertical. From the principle of linear superposition, the convolution of the two kernels with the 2-dimension image is identical to the original 2-dimensional convolution. Using a separable kernel requires only $2k$ multiplication and addition operations per point, a significant improvement over the 2-dimensional operator.

Consider the example of a 5×5 truncated Gaussian kernel:

$$2d\ kernel = \frac{1}{256}\begin{bmatrix} 1 & 4 & 6 & 4 & 1 \\ 4 & 16 & 24 & 16 & 4 \\ 6 & 24 & 36 & 24 & 6 \\ 4 & 16 & 24 & 16 & 4 \\ 1 & 4 & 6 & 4 & 1 \end{bmatrix}$$

The Gaussian kernel is the only linearly separable kernel which is also **radially symmetric**. This kernel can be separated into two 1-dimensional kernels, horizontal vertical with approximate values: $[1, 4, 6, 4, 1]$. As a result, the equivalent to the computationally complex 2-d filter can be applied by two passes (vertical and horizontal) of the computationally efficient 1-d filters.

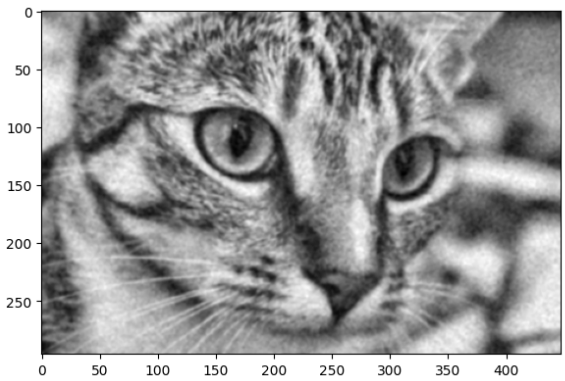
> **Exercise 2-9:** You will now apply the separable 1-d kernels to the noisy gray-scale cat image by the following steps:

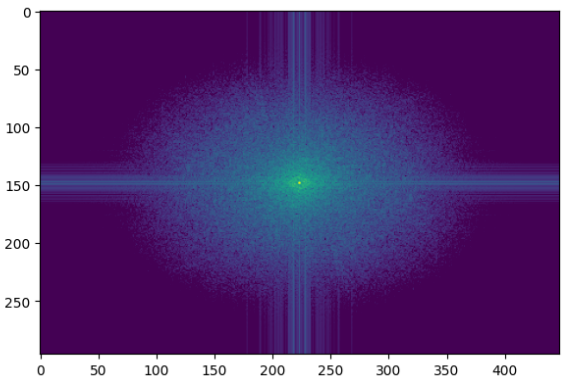
1. First, you will verify that the separable kernels are indeed a good approximation of the 2-d kernel. Compute the outer product of the 1-d kernel with itself. Display this result, along with the nomalizaiton constant, which you can compute as `kernel_norm = np.sum(d2_kernel)`.
2. In the next code cell create a function that applies the 1-d kernel to the rows of the image. Your function should have two arguments, the image and the 1-d kernel, and will return the result. Keep in mind that the number of columns of the result will be k points less. You the 1-d convolution function you created for Exercise 2-2. Remember to divide the final resul by the normalization constant.
3. Apply your function to the gray-scale cat image with added Gaussian noise. Then apply your function with the same kernel to the transpose of the image.
4. Display the filtered image and is discrete Fourier transform side by side as you did for Exercise 2-5.

In [ ]:
```
## Put your code below

# 1.
# Outer product of the 1D kernel with itself
kernel1 = np.array([1, 4, 6, 4, 1])
outer_product = np.outer(kernel1, kernel1)
print('\nOuter product for 1D kernel:')
print(outer_product)

kernel2 = np.array([[1, 4, 6, 4, 1], [4, 16, 24, 16, 4], [6, 24, 36, 24, 6],

kernel_norm = np.sum(kernel2)
print(f'\nNormalization constant: {kernel_norm}\n\n')
```

```
Outer product for 1D kernel:
[[ 1  4  6  4  1]
 [ 4 16 24 16  4]
 [ 6 24 36 24  6]
 [ 4 16 24 16  4]
 [ 1  4  6  4  1]]


2D kernel:
[[ 1  4  6  4  1]
 [ 4 16 24 16  4]
 [ 6 24 36 24  6]
 [ 4 16 24 16  4]
 [ 1  4  6  4  1]]

Normalization constant: 256
```

In [ ]:
```
## Complete the function
def separate_convolve(img, kernel):
    height, _ = img.shape

    result = []
    [result.append(conv_1d(img[i], kernel)) for i in range(height)]
    result = np.array(result)

    return result


kernel = kernel1

cat_separated_filtered = separate_convolve(cat_grayscale_equalized_gaussian,
cat_separated_filtered = np.transpose(separate_convolve(np.transpose(cat_sep

fig, ax = plt.subplots(1,2, figsize=(15, 20))
_ = ax[0].imshow(cat_separated_filtered, cmap = plt.get_cmap('gray'))
_ = ax[1].imshow(np.square(np.abs(fft.fftshift(fft.fft2(cat_separated_filter
```





> 1. Does the outer product of the 1-d kernel with itself represent a good approximation of the truncated 2-d kernel?
> 2. Compare the filtered image and Fourier transform of the filtered image to those of the unfiltered noisy image. What evidence can you see that the filter reduced the noise at the expense of blurring the image?
>
> **End of exercise.**

> **Answers:**

> 1. Yes, it does.
> 2. The filtered image is blurred, compared with the noisy version. The FFT of the filtered image is less spread out, compared with the noisy version. This is evidence that the filter reduced the noise at the expense of blurring the image.

Image Pyramids

We have explored the effect of changing filter bandwidth on images. This approach is effective if an image has a limited range of scales of interest. However, in many cases, images contain features of interest at many scales. To address these situations we need a general method for decomposing images into multiple scales. The *f*image pyramid** is just such a technique.

An image pyramid uses Gaussian filters to down-sample the image by **octaves** or powers or 2; $[1/2, 1/4, 1/8, \ldots]$. This process creates a hierarchy of scales from an original image, the image pyramid. The pyramid is a data structure is therefore an convenient method to represent multiple scales of an image.

The pyramid is created by recursively Gaussian filtering the image and then sub-sampling by a factor of 2. In summary, the recursion works by starting with the original image and subsequently the last down-sample image to filter and down-sample. The down-sampling terminates either at a predetermined image size or when the resampled image is 1×1.

The code in the cell below creates an image pyramid for the cat image. First, the cat image is down-sampled to an even binary size. Then the image pyramid is constructed. Execute this code.

> **Note:** It is most convenient to perform down-sampling by octaves on an image with even binary number dimensions. In fact, the skimage.transform.pyramid_gaussian can only work with images of even binary dimensions and will raise an exception otherwise.

In [ ]:
```
print(cat_image.shape)
cat_image_resize = transform.resize(cat_image, (256,256))
rows, cols, dim = cat_image_resize.shape
cat_pyramid = tuple(pyramid_gaussian(cat_image_resize, downscale=2, channel_
```

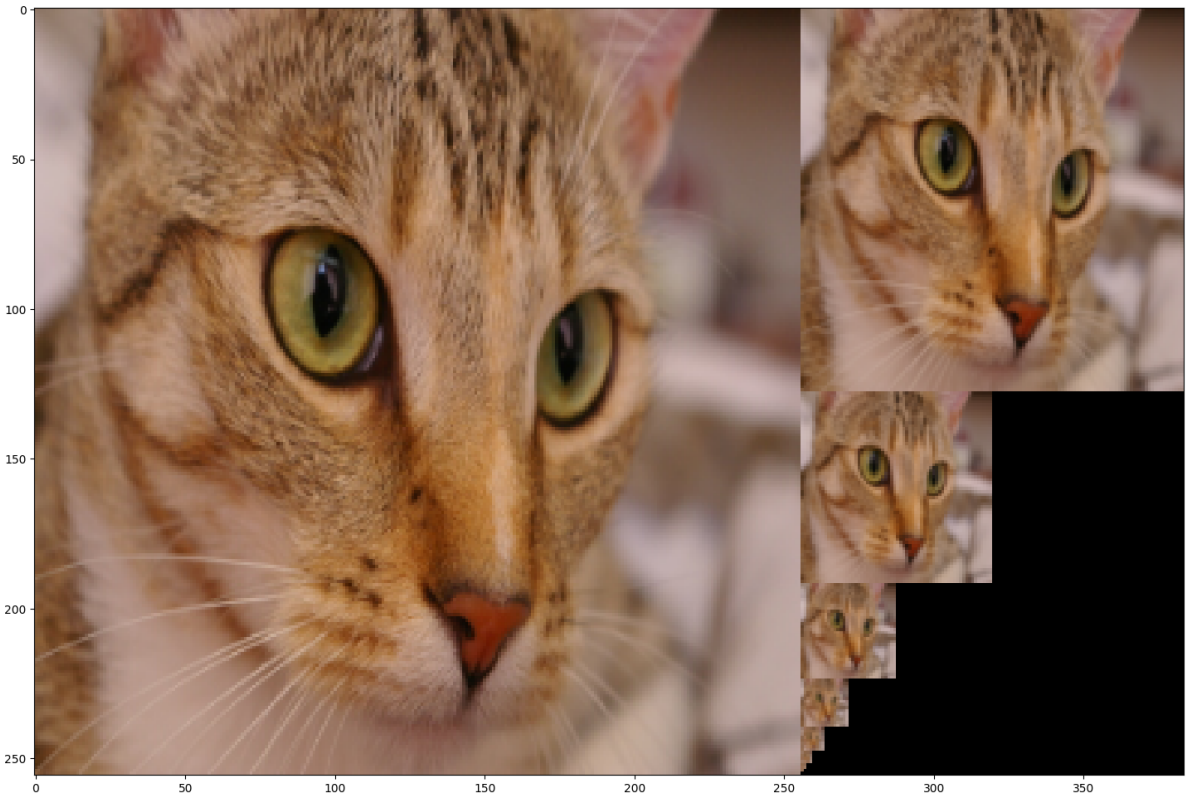
```
(300, 451, 3)
```

What does this image pyramid look like? To find out, execute the code in the cell below to print the dimensions of the down-sampled images and display them.

In [ ]:
```
composite_image = np.zeros((rows, cols + cols // 2, 3), dtype=np.double)

composite_image[:rows, :cols, :] = cat_pyramid[0]

i_row = 0
for p in cat_pyramid[1:]:
    n_rows, n_cols = p.shape[:2]
    print('Shape of the image in pyramid = ', str(p.shape[:2]))
    composite_image[i_row:i_row + n_rows, cols:cols + n_cols] = p
```

```
    i_row += n_rows

fig, ax = plt.subplots(figsize=(18, 12))
ax.imshow(composite_image)
plt.show()
```

```
Shape of the image in pyramid =  (128, 128)
Shape of the image in pyramid =  (64, 64)
Shape of the image in pyramid =  (32, 32)
Shape of the image in pyramid =  (16, 16)
Shape of the image in pyramid =  (8, 8)
Shape of the image in pyramid =  (4, 4)
Shape of the image in pyramid =  (2, 2)
Shape of the image in pyramid =  (1, 1)
```



> **Exercise 2-10:** Notice the dimensions of the down-sampled images along the pyramid. How does the resolution change along the members of the image pyramid relate to the scale of the visible image features?

> **Answer:**

> This pyramid shows the decimation of the image where features are lost as the image is down-sampled. As expected, this feature loss starts with the smallest features in the previous level. At the same time, the resolution degrades to the point where the image is no longer recognizable.

In [ ]: